```c
// Computer Program Listing Appendix Under 37 CFR 1.52(e)
// patent.c
// Copyright (c) 2004. Sybase, Inc. All Rights Reserved.
/*
** Define the maximumn total number of samples to take for each statistics
** collection. This is used for tuning the degree of parallelism for
** concurrent database recovery.
*/
#define TOTAL_NUM_SAMPLES 3
/*
** Information for a database to be recovered.
** The database items to be recovered are in the order to be recovered as an
** array in REC_ORDER_INFO
*/
typedef struct rec_order_item
{
 dbid_t  dbid;
 int      status;  /* status of each recovery item. */
 struct pss *recovery_pss; /* the process id of the thread that is
     ** recovering this database.
     */
 int  onlmsg;  /* store the online message that
     ** will be printed when the database
     ** is onlined in a deferred mode.
     ** (If the database is onlined without
     ** any delay, the message will be
     ** printed right away, and therefore
     ** we don't need to store it.)
     */
 int  sample_result[TOTAL_NUM_SAMPLES];
     /*
     ** Store the statistics collected
     ** each time a thread is spawned.
     **
     ** Although this field is for each
     ** spawned thread, to make the code
     ** simpler, we use the array of
     ** recover order items to store this
     ** information. This array should
     ** have enough members to hold the
     ** information, because the number
     ** of spawned threads cannot be
     ** more than the number of recovery
     ** items.
     */
} REC_ORDER_ITEM;
#define REC_ORDER_ITEM_SIZE   sizeof(REC_ORDER_ITEM)
/*
** Defines for status field in rec_order_item.
** WARNING: If you add/change a #define here, please make the
** corresponding change to the string definitions for mnemonics for
** rec_order_item->status.
*/
/*
** status indicating item has no more recovery work to do. or'ed in
** REC_ITEM_DONE.
** WARNING: If you add/change a status bit which indicates that the
** recovery work on an item is done, please make corresponding change in
** mask REC_ITEM_DONE.
*/
#define REC_ITEM_SKIP_RECOVERY  0x00000001
    /* this indicates that this db
    ** will not be recovered even though
    ** its item is in the rec_order_item
    ** array. recovery will skip this
    ** db.
    */
#define REC_ITEM_RECOVERED  0x00000002
    /*
    ** This database is recovered
    */
#define REC_ITEM_FAILED   0x00000004
    /*
    ** Recovery of this item has failed.
    */
/*
** Status regarding the recovery state of an item. Or'ed in
** REC_ITEM_RECOVERY_STATE.
** WARNING: If you add/change a status bit regarding the recovery
** state of a recovery item (database), please make corresponding change
** in the mask REC_ITEM_RECOVERY_STATE.
*/
#define REC_ITEM_NOT_RECOVERED  0x00000008
    /*
    ** This database is not yet recovered
    */
#define REC_ITEM_RECOVERING  0x00000010
    /*
    ** This database is currently being
    ** recovered
    */
#define REC_ITEM_ONL_IMMEDIATELY 0x00000020
    /*
    ** this item can be brought online
    ** without any checking.
    ** This status is set when none of
    ** the rec_order_items before this item
    ** has strict online order specified.
    */
#define REC_ITEM_ONL_WITH_STRICT_ORDER 0x00000040
```

```
    /*
    ** this item has "strict" online
    ** order specified.
    */
#define REC_ITEM_WAITING_TO_ONLINE 0x00000080
    /*
    ** this item could not be brought
    ** online immediately because
    ** of some strict online order
    ** violation.
    */
#define REC_ITEM_TO_BE_ONLINED  0x00000100
    /*
    ** this item can now be onlined
    ** because the online order conflict
    ** that used to block it from onlining
    ** is no longer true.
    */
/*
** Other status.
*/
#define REC_ITEM_USER_TEMPDB  0x00000200
    /*
    ** this bit is set when the database
    ** is a user defined tempdb.
    */
/* String definitions for mnemonics for rec_order_item->status */
# define REC_ORDER_ITEM_STAT_BIT00_STR  "REC_ITEM_SKIP_RECOVERY"
# define REC_ORDER_ITEM_STAT_BIT01_STR  "REC_ITEM_RECOVERED"
# define REC_ORDER_ITEM_STAT_BIT02_STR  "REC_ITEM_FAILED"
# define REC_ORDER_ITEM_STAT_BIT03_STR  "REC_ITEM_NOT_RECOVERED"
# define REC_ORDER_ITEM_STAT_BIT04_STR  "REC_ITEM_RECOVERING"
# define REC_ORDER_ITEM_STAT_BIT05_STR  "REC_ITEM_ONL_IMMEDIATELY"
# define REC_ORDER_ITEM_STAT_BIT06_STR  "REC_ITEM_ONL_WITH_STRICT_ORDER"
# define REC_ORDER_ITEM_STAT_BIT07_STR  "REC_ITEM_WAITING_TO_ONLINE"
# define REC_ORDER_ITEM_STAT_BIT08_STR  "REC_ITEM_TO_BE_ONLINED"
# define REC_ORDER_ITEM_STAT_BIT09_STR  "REC_ITEM_USER_TEMPDB"
# define REC_ORDER_ITEM_STAT_BIT10_STR  Statbit_unused_str
# define REC_ORDER_ITEM_STAT_BIT11_STR  Statbit_unused_str
# define REC_ORDER_ITEM_STAT_BIT12_STR  Statbit_unused_str
# define REC_ORDER_ITEM_STAT_BIT13_STR  Statbit_unused_str
# define REC_ORDER_ITEM_STAT_BIT14_STR  Statbit_unused_str
# define REC_ORDER_ITEM_STAT_BIT15_STR  Statbit_unused_str
# define REC_ORDER_ITEM_STAT_BIT16_STR  Statbit_unused_str
# define REC_ORDER_ITEM_STAT_BIT17_STR  Statbit_unused_str
# define REC_ORDER_ITEM_STAT_BIT18_STR  Statbit_unused_str
# define REC_ORDER_ITEM_STAT_BIT19_STR  Statbit_unused_str
# define REC_ORDER_ITEM_STAT_BIT20_STR  Statbit_unused_str
# define REC_ORDER_ITEM_STAT_BIT21_STR  Statbit_unused_str
# define REC_ORDER_ITEM_STAT_BIT22_STR  Statbit_unused_str
# define REC_ORDER_ITEM_STAT_BIT23_STR  Statbit_unused_str
# define REC_ORDER_ITEM_STAT_BIT24_STR  Statbit_unused_str
# define REC_ORDER_ITEM_STAT_BIT25_STR  Statbit_unused_str
# define REC_ORDER_ITEM_STAT_BIT26_STR  Statbit_unused_str
# define REC_ORDER_ITEM_STAT_BIT27_STR  Statbit_unused_str
# define REC_ORDER_ITEM_STAT_BIT28_STR  Statbit_unused_str
# define REC_ORDER_ITEM_STAT_BIT29_STR  Statbit_unused_str
# define REC_ORDER_ITEM_STAT_BIT30_STR  Statbit_unused_str
# define REC_ORDER_ITEM_STAT_BIT31_STR  Statbit_unused_str
/* Mask indicating that the work on the item has been completed. */
# define REC_ITEM_DONE (REC_ITEM_SKIP_RECOVERY | REC_ITEM_RECOVERED | \
   REC_ITEM_FAILED)
/*
** Mask containing all status bits related to the recovery item's recovery
** state. These status bits need to be cleaned up first in either of the
** following cases:
** 1. When an item is cleaned up, before we set the item to the
**    REC_ITEM_RECOVERED.
** 2. before an item is to be marked as REC_ITEM_SKIP_RECOVERY.
*/
# define REC_ITEM_RECOVERY_STATE     \
 (REC_ITEM_NOT_RECOVERED | REC_ITEM_RECOVERING |   \
 REC_ITEM_ONL_IMMEDIATELY | REC_ITEM_ONL_WITH_STRICT_ORDER | \
 REC_ITEM_WAITING_TO_ONLINE | REC_ITEM_TO_BE_ONLINED)
typedef struct rec_order_info
{
 int  rec_info_memsize;
    /*
    ** the size of the chunk of memory
    ** we allocated. This is needed
    ** for exception handler to free
    ** the memory.
    */
 int  rec_order_size; /*
    ** the number of db items that are in
    ** the rec_order_info structure.
    ** Note that this number could be
    ** bigger than num_dbs_to_recover because
    ** there could be databases that will
    ** skip recovery.
    */
 int            num_dbs_to_recover;
    /*
    ** the number of dbs that actually
    ** need to be recovered.
    */
 int  num_dbs_done;
    /*
    ** number of databases that have
    ** completed the recovery process
    */
 int  read_counter; /*
    ** store the statistics collected
    ** during each sample period.
    */
 int  num_rec_threads;
    /* number of recovery threads running,
    ** including the frozen threads.
    */
 int  optimal_num_rec_threads;
    /* the optimal number of recovery
    ** threads that the server will run
    ** with.
    */
 int  first_offline_strict_order_item;
    /*
    ** the rec_order_item of the first
    ** strict order item which is still
    ** offline.
    */
 kpid_t  latest_spawned_thread_kpid;
 dbid_t  metadbid; /* the dbid of the meta database. */
 REC_ORDER_ITEM  *rec_order_item;
    /*
    ** Start of an array of recovery items.
    */
}REC_ORDER_INFO;
#define REC_ORDER_INFO_SIZE   sizeof(REC_ORDER_INFO)
/*
** defines for status field in recovery_info
** WARNING: If you add/change a #define here, please make the
** corresponding change to the string definitions for
** mnemonics for recovery_info.status.
*/
#define REC_INFO_INIT   0x00000000
#define REC_INFO_PARALLEL  0x00000001
     /* indicates that user db
     ** recovery by recovery
     ** threads will start. The
     ** initial thread will be put
     ** to sleep on this status
     ** until the recovery completes.
     */
#define REC_INFO_TUNE_COMPLETE  0x00000002
     /* indicates that recovery
     ** tuning has completed.
     */
#define REC_INFO_COLLECT_STAT  0x00000004
     /* indicates that sample
     ** statistics need to be
    ** collected for recovery.
    */
#define REC_INFO_BOOTTIME_RECOVERY 0x00000008
#define REC_INFO_FAILOVER_RECOVERY 0x00000010
#define REC_INFO_TOO_MANY_THREADS 0x00000020
    /*
    ** indicates that the tuning
    ** thread has found that
    ** there are more threads
    ** running than the system
    ** could handle. With this
    ** status set, the system
    ** will reduce the number
    ** of running recovery threads
    ** by either putting one of
    ** the threads to sleep on this
    ** status or letting one of
    ** the threads exit.
    */
#define REC_INFO_GOBACK_TO_SERIAL  0x00000040
    /* indicating that user has
    ** requested the server to
    ** go back to serial recovery
    ** by changing the config
    ** parameter value to 1.
    */
#define REC_INFO_INVALID_STAT   0x00000080
    /* indicating that the current
    ** statisitcs is not valid
    ** because the latest spawned
    ** thread had entered redo
    ** phase during the stat
    ** collection period.
    */
#define REC_INFO_FAILOVER_FAIL   0x00000100
    /*
    ** One of the databases failed
    ** to recover during HA failover
    */
#define REC_INFO_CACHE_NEEDRESTORE  0x00000200
    /*
    ** Indicate that
    ** the default data cache
    ** was tuned by the recovery
    ** process. Thus, recovery
    ** process needs to restore
    ** the old values aftetwards.
    */
#define REC_INFO_BY_DBID   0x00000400
    /*
```

```
     ** Indicating that the server
     ** will recover databases
     ** by their dbid order
     ** because it failed to
     ** build the recovery items
     ** structure in memory.
     */
#define REC_INFO_USE_LARGEST_IO_POOL  0x00000800
     /*
     ** Indicating that recovery
     ** will use the largest io
     ** pool.
     */
/* String definitions for mnemonics for recovery_info.status */
# define RECOVERY_INFO_STAT_BIT00_STR  "REC_INFO_PARALLEL"
# define RECOVERY_INFO_STAT_BIT01_STR  "REC_INFO_TUNE_COMPLETE"
# define RECOVERY_INFO_STAT_BIT02_STR  "REC_INFO_COLLECT_STAT"
# define RECOVERY_INFO_STAT_BIT03_STR  "REC_INFO_BOOTTIME_RECOVERY"
# define RECOVERY_INFO_STAT_BIT04_STR  "REC_INFO_FAILOVER_RECOVERY"
# define RECOVERY_INFO_STAT_BIT05_STR  "REC_INFO_TOO_MANY_THREADS"
# define RECOVERY_INFO_STAT_BIT06_STR  "REC_INFO_GOBACK_TO_SERIAL"
# define RECOVERY_INFO_STAT_BIT07_STR  "REC_INFO_INVALID_STAT"
# define RECOVERY_INFO_STAT_BIT08_STR  "REC_INFO_FAILOVER_FAIL"
# define RECOVERY_INFO_STAT_BIT09_STR  "REC_INFO_CACHE_NEEDRESTORE"
# define RECOVERY_INFO_STAT_BIT10_STR  "REC_INFO_BY_DBID"
# define RECOVERY_INFO_STAT_BIT11_STR  "REC_INFO_USE_LARGEST_IO_POOL"
# define RECOVERY_INFO_STAT_BIT12_STR  Statbit_unused_str
# define RECOVERY_INFO_STAT_BIT13_STR  Statbit_unused_str
# define RECOVERY_INFO_STAT_BIT14_STR  Statbit_unused_str
# define RECOVERY_INFO_STAT_BIT15_STR  Statbit_unused_str
# define RECOVERY_INFO_STAT_BIT16_STR  Statbit_unused_str
# define RECOVERY_INFO_STAT_BIT17_STR  Statbit_unused_str
# define RECOVERY_INFO_STAT_BIT18_STR  Statbit_unused_str
# define RECOVERY_INFO_STAT_BIT19_STR  Statbit_unused_str
# define RECOVERY_INFO_STAT_BIT20_STR  Statbit_unused_str
# define RECOVERY_INFO_STAT_BIT21_STR  Statbit_unused_str
# define RECOVERY_INFO_STAT_BIT22_STR  Statbit_unused_str
# define RECOVERY_INFO_STAT_BIT23_STR  Statbit_unused_str
# define RECOVERY_INFO_STAT_BIT24_STR  Statbit_unused_str
# define RECOVERY_INFO_STAT_BIT25_STR  Statbit_unused_str
# define RECOVERY_INFO_STAT_BIT26_STR  Statbit_unused_str
# define RECOVERY_INFO_STAT_BIT27_STR  Statbit_unused_str
# define RECOVERY_INFO_STAT_BIT28_STR  Statbit_unused_str
# define RECOVERY_INFO_STAT_BIT29_STR  Statbit_unused_str
# define RECOVERY_INFO_STAT_BIT30_STR  Statbit_unused_str
# define RECOVERY_INFO_STAT_BIT31_STR  Statbit_unused_str
#define IS_FAILOVER_THREAD() (Resource->rrecovery_info.status \
    & REC_INFO_FAILOVER_RECOVERY)
#define REC_ORDER_INIT_STRUCT(attrinfo)\
 attrib_initstruct(&attrinfo); \
 attrinfo.aiclass = HA_CLASS; \
 strncpy((char *)&attrinfo.aitype, (char *)ATTR_TYPE_DATABASE, 2);\
 attrinfo.aiattrib = HA_DATABASE_RECOVERY_ORDER
/*
** Define the databases which will not be recovered:
** 1. database is in load OR
** 2. database is marked bypass OR
** 3. database is user created tempdb AND server is doing failover recovery
*/
#define DB_SKIP_RECOVERY(datstat, datstat3) \
 (((datstat) & DBT_INLDDB) || ((datstat) & DBT_USE_NOTREC) || \
  (((datstat3) & DBT3_USER_TEMPDB) && IS_FAILOVER_THREAD()) )
/*
** This context structure is common among several recovery core
** functions. It is used to track resources acquired by recovery,
** so that these resources may be released on backing out.
*/
typedef struct
{
 struct itag *itagp;
 struct sdes *logsdes;
 struct xdes *xdes;
 XTABLE  *xtable;
 struct vbitmap *allocbitmap;
 RECTABLE *rectable;
 REC_FP_TABLE *fptab;
 struct pss *pss;
 BYTE  *rowbuf;
 int  check_freespace;
 int  xdes_endstat;
} REC_BKOUT_CTX;
/*
** Defined the backout structure for the callers to recovery. This structure
** and its cleanup function will be registered in pss->pbkout_rec_caller.
*/
typedef struct
{
 struct dbtable  *backout_dbt; /*
      ** dbtable opened for the
      ** database being recovered.
      */
 struct dbtable  *master_dbt; /*
      ** master database opened
      ** at beginning of dorecover
      */
 struct sdes  *db_sdes;       /* store the sysdatabases
      ** opened in the following
      ** places:
      ** 1. populate the
      ** rec_order_info struct,
    ** 2. recovery by dbid order,
    ** 3. reset some status bits
    ** in sysdatabases after
    ** recovery.
    */
 struct rec_order_info *rec_order_info;
 int   rec_info_memsize;
    /* size of memory allocated
    ** for rec_order_info.
    */
 int   rec_order_num; /*
    ** the number of the item
    ** being recovered.
    */
 dbid_t   curdbid; /*
    ** dbid of the database being
    ** recovered.
    */
 SYB_BOOLEAN  model_locked;
 struct dbtable  *modeldbt;
 SYB_BOOLEAN  global_cleanup; /* This boolean will determine
    ** if rec__caller_hdlr() needs
    ** to call the clean up func
    ** in pss before return.
    */
 struct dbtable  *oldmaster_dbt; /*
    ** Only filled during failover
    ** recovery: dbtable for
    ** master_companion
    */
 struct sdes  *sysdbs;   /*
    ** Only filled during failover
    ** recovery: sysdatabases
    ** opened during failover.
    */
} REC_CALLER_BKOUT_CTX;
/*
** This structure contains the information that will be passed to the
** spawned recovery tasks by the initial recovery thread.
*/
typedef struct rec_caller_arg
{
 SYB_BOOLEAN  model_recovered;
    /* user defined tempdbs will
    ** only be recovered (i.e.
    ** recreated) if model database
    ** has been recovered.
    */
 engid_t   engine_num; /* This is used to affiliate
    ** recovery threads to
```

```
     ** the engines under trace
     ** flag.
     */
 struct dbtable  *modeldbt; /* This will be used
     ** to recover user defined
     ** tempdbs.
     */
} REC_CALLER_ARG;
/*
** macros to access the recovery resource registry for cleanup.
*/
#define RECBKOUT_CTX_FROM_PSS(pss) (pss)->pbkout_rec.ptaskdata
/*
** macros to access the resource registry for cleanup for boot
*/
#define REC_CALLER_CTX_FROM_PSS(pss) (pss)->pbkout_rec_caller.ptaskdata
/*
** Define the rec_mode that could be passed in to rec__boot_recover_dbs
** and rec_failover_recover_dbs.
*/
#define READ_FAILED 1 /* failed to populate the rec_order_info
    ** structure, and therefore have to
    ** recovery in serial by dbid order.
    */
#define SERIAL  2 /* recover in serial using the populated
    ** rec_order_info structure.
    */
#define PARALLEL 3 /* recover in parallel using the populated
    ** rec_order_info structure.
    */
/* Return values defined for rec_getnextdb_to_recover() and
** rec_getnextdb_by_dbid(). Only the first 3 defines will be used
** by rec_getnextdb_by_dbid().
*/
# define REC_GOT_NOMORE_DB  0 /* no more db to recover */
# define REC_GOT_NEXT_DB  1 /* found a non-tempdb database to
       ** recover */
# define REC_GOT_TEMPDB   2
     /* found a tempdb and will go ahead and
     ** recover it.
     */
# define REC_ONLINE_DB   3
# define REC_NEED_TO_EXIT  4 /* the thread will exit */
# define REC_WAKEUP_FROZEN_THREAD 5
     /* caller will wake up the frozen
     ** thread and then exit.
     */
/*
** Return values defined for rec__collect_statistics() and
** rec__stat_degraded()
```

```
*/
# define REC_RECOVERY_COMPLETE  -1
    /* This is returned when recovery was
    ** found to have completed during the
    ** statistics collection.
    */
# define REC_INVALID_STAT  -2
    /* this statistics is invalid.
    ** This is returned when the latest
    ** spawned thread completed the
    ** analysis pass before sample period
    ** ended.
    */
# define REC_STAT_DEGRADATION  0
    /*
    ** This is returned when
    ** the current statistics has
    ** showed unacceptable degradation
    ** comparing to the prev statistics.
    */
# define REC_STAT_NO_DEGRADATION 1
    /* This is returned when
    ** the current statistics has NOT
    ** showed unacceptable degradation
    ** comparing to the prev statistics.
    */
/* Passed in parameters to rec__tune_bufpools() */
# define REC_SET_CONFIG  0 /* tune the buffer pool configurations
     ** according to recovery's needs */
# define REC_RESTORE_CONFIG 1 /* restore the old configuration before the
     ** recovery change */
/*
** The values indicating that the fields in the buffer pools are not changed.
** Cannot use -1, because APF_UNSPECIFIED_OR_DEFAULT, which is a valid old
** apf value, is defined as -1.
*/
# define VALUE_NOT_CHANGED -2
/* Macro to test is no tuning was done to the passed in buffer pool. */
# define REC_POOL_NOT_TUNED(pool)     \
 (!((pool)->bstatus & BPOOL_SIZE_CHANGED) && \
  ((pool)->bold_apf_percent == VALUE_NOT_CHANGED))
/* Heuristics defined for some tuning factors. */
# define REC_TUNE_POOL_RATE 0.4 /*
    ** This will guide the tuning of buffer
    ** pools in the default data cache
    ** during recovery:
    **
    ** size_of_largest_mass_pool =
    **  REC_TUNE_POOL_RATE *
    **  (size_of_default_pool +
    **   size_of_largest_mass_pool)
    **
    ** This heuristic value is determined
    ** after some benchmark testing using
    ** in-house machines and high
    ** end SUN system which represents
    ** what high end customers are using.
    */
# define REC_OPTIMAL_APF 80 /*
    ** This defines the optimal async
    ** prefetch limit for recovery. This
    ** value will be set for the buffer
    ** pools that will be used for recovery.
    ** And the old value will be restored
    ** at the end of recovery.
    */
# define MAX_PERFORMANCE_DROP_ALLOWED 0.25
    /*
    ** This defines the maximum allowable
    ** performance drop between the new
    ** the previous statistics while the
    ** tuning thread is collecting
    ** statistics as recovery threads
    ** are being spawned.
    **
    ** During testing, it is found that
    ** when the requests from recovery
    ** reaches the capacity of the
    ** I/O subsystem, the statistics
    ** would drop dramatically (more than
    ** 50%). The reason why we allow
    ** a little performance drop is to
    ** filter out the noise of performance
    ** variations which are not related to
    ** the I/O subsystem capacity.
    */
# define MAX_DIFF_RATE_ALLOWED_BETWEEN_SAMPLES 0.15
/*
** This macro is used to test if the passed in pss is the thread that tuning
** process is inspecting.
** It returns TRUE under the following conditions:
** 1. the server is in tuning process AND
** 2. this thread is the thread that was just spawned.
**
** If the thread that is under inspection enters any of the following
** paths, the current sample will be considered invalid:
** 1. if it is outside the acceptable zone for sampling. The acceptable zone is
**    the logbounds phase where log pages are read
**    This means that the database doesn't have a big enough recoverable log
**    and therefore the current sample cannot be used to reflect the I/O
**    subsystem's performance.
** 2. if it enters the recovery function for tempdb. Since the recovery of
**    tempdb is quite different than that of normal dbs (tempdb recovery is
**    just the recreation of the tempdb), we cannot use the sample collected
**    during tempdb recovery. User tempdbs will be skipped during recovery
**    tuning period, but if there is no more normal user dbs to recover,
**    recovery threads will pick up tempdbs even during tuning period. Thus,
**    we need this rule.
** 3. if it enters the cleanup function to be cleaned up.
**
** Synchronization:
**  Spinlock needs to be held before calling this macro.
*/
# define REC_THREAD_UNDER_INSPECTION(pss)    \
 (!(Resource->rrecovery_info.status & REC_INFO_TUNE_COMPLETE) && \
  (Resource->rrecovery_info.rec_order_info) &&   \
  (Resource->rrecovery_info.rec_order_info->latest_spawned_thread_kpid \
   == (pss)->pkspid))
/*
** This macro is used to determine if a database needs to be checked
** for online order conflict before it is made accessible.
**
** The database needs to be checked if all of the following hold
** true:
** 1. The rec_order_num of the item representing the database is not
** UNUSED.
** 2. AND the recovery item representing the database does not have
** REC_ITEM_ONL_IMMEDIATELY set. This bit is set for databases which
** can be brought online immediately.
** 3. AND server is in the middle of parallel recovery.
**
** Note on synchronization:
** The macro is called under spinlock.
*/
# define ONL_ORDER_CHECK_NEEDED(rec_order_num, rec_order_info)  \
  (((rec_order_num) != UNUSED) && (rec_order_info) &&  \
   (Resource->rrecovery_info.status & REC_INFO_PARALLEL) && \
   !(rec_order_info->rec_order_item[(rec_order_num)].status & \
     REC_ITEM_ONL_IMMEDIATELY))
/*
** ANSI-C recondite rule (K&R second edition page 213)
** See generic/kinclude/ksrc_dcl.h for details.
*/
struct recovery_info;
/* Prototype for parallel recovery */
int  rec_build_recovery_info  PROTO((REC_CALLER_BKOUT_CTX *,
       struct dbtable *));
void  rec_run_parallel_recovery PROTO((REC_CALLER_BKOUT_CTX *,
       REC_CALLER_ARG *));
int  rec_getnextdb_by_dbid  PROTO((REC_CALLER_BKOUT_CTX *,
        dbid_t *, dbid_t));
int  rec_getnextdb_to_recover PROTO((REC_CALLER_BKOUT_CTX *));
SYB_BOOLEAN rec_online_order_conflict PROTO((int));
void  rec_freeze_thread  PROTO((void));
void  rec_config_param_modify  PROTO((int16));
int  rec_caller_backout  PROTO((void));
void  rec_cleanup_recovery_item PROTO((int, SYB_BOOLEAN));
void  rec_makedb_accessible  PROTO((void));
int  rec_failover_recover_dbs PROTO((REC_CALLER_BKOUT_CTX *,
         int));
void  prRECOVERY_INFO   PROTO((struct recovery_info *));
SYB_BOOLEAN rec_thread_cleanup  PROTO((FNIPARAM, SYB_BOOLEAN));
int  rec_cmp_marker   PROTO((struct sdes *,
        struct xlrmarker *,
        struct xlrmarker *));
/*
** REC__SET_NEXT_STRICT_ORDER_ITEM
**
** Purpose:
** This function is called because the first strict order item in
** the Resource->rrecovery_info.rec_order_info needs to be reset.
** It can be called in the following places:
** 1. during rec_build_recovery_info() when the
**    first_offline_strict_order_item will skip recovery
** 2. during rec_cleanup_recovery_item() when the
**    first_offline_strict_order_item is online.
**
** It does the following:
** 1. Search from the passed in rec_order_num to the passed in
**    end_of_list to find the next item with strict online order;
**    If found, set the first_offline_strict_order_item to its order
**    number.
** 2. While we are searching through the list:
**  Set REC_ITEM_ONL_IMMEDIATELY in all items that are not done
**  with recovery and are between the old and new first strict
**  order items.
**
**   These items didn't have REC_ITEM_ONL_IMMEDIATELY bit set
**  before because the old first strict order item was not onlined,
**  but now since it is online, these items can be brought online
**  right away without checking for online ordering.
**
**  Use the following item sequence as an example of what this function
** does:
**  (S represent items with strict recovery order,
**  NoS represent items without strict recovery order,
**  and the number within braces is the item number):
**  item sequence:
**    NoS{1}, S{2}, NoS{3}, NoS{4}, S{5}, NoS{6}
**  After rec_build_recovery_info(), the status related to online ordering:
```

```
**  #1: REC_ITEM_ONL_IMMEDIATELY
**  #2: REC_ITEM_ONL_WITH_STRICT_ORDER
**  #3 and #4: None.
**  #5: REC_ITEM_ONL_WITH_STRICT_ORDER
**  #6: None.
**  And the first_offline_strict_order_item in rec_order_info is 2.
**
**  So except for item #1, all other items need to go through
**  rec_online_order_conflict() before online.
**
**  When recovery item #2 has been onlined, rec_cleanup_recovery_item() will
**  call rec__set_next_strict_order_item(), after which, the items status
**  will be:
**  #1: nothing.
**  #2: nothing.
**  #3  and #4: set REC_ITEM_ONL_IMMEDIATELY
**  #5: REC_ITEM_ONL_WITH_STRICT_ORDER and first_offline_strict_order_item
**     will be set to 5
**  #6: nothing.
**
** Parameter:
** rec_order_num - the current first_offline_strict_order_item. This
**   is the start of the search.
** end_of_list - The passed in end of search.
** is_in_cleanup - TRUE if called by rec_cleanup_recovery_item.
**   FALSE otherwise.
**
** Return:
** None.
**
** Synchronization:
** If this is called by rec_cleanup_recovery_item(), spinlock must
** have been held before going into this function.
** If called by rec_build_recovery_info(), no spinlock is needed.
**
** History:
** 1/2003 (fzhou) - created.
*/
SYB_STATIC void
rec__set_next_strict_order_item(int rec_order_num, int end_of_list,
   SYB_BOOLEAN is_in_cleanup)
{
 int  count;
 int  *first_strict_order;
 REC_ORDER_INFO *rec_order_info;
 REC_ORDER_ITEM *temp_item;
 rec_order_info = Resource->rrecovery_info.rec_order_info;
 first_strict_order = &rec_order_info->first_offline_strict_order_item;
 if (is_in_cleanup)
 {
  SPINLOCKHELD(Resource->ha_spin);
 }
 for (count = rec_order_num + 1; count <= end_of_list; count++)
 {
  temp_item = &rec_order_info->rec_order_item[count];
  /*
  ** If we find the next recovery item with strict
  ** online order, point the first_strict_order to
  ** this item.
  */
  if (temp_item->status & REC_ITEM_ONL_WITH_STRICT_ORDER)
  {
   *first_strict_order = count;
   /*
   ** We have found the next strict order item,
   ** no need to go any further.
   */
   break;
  }
  else if (!(temp_item->status & REC_ITEM_DONE))
  {
   /*
   ** Before we find the next strict order item,
   ** set REC_ITEM_ONL_IMMEDIATELY bit in the
   ** items which,
   ** 1. don't have a strict recovery order, AND
   ** 2. still have some recovery work to do.
   */
   SYB_ASSERT(!(temp_item->status &
     REC_ITEM_ONL_WITH_STRICT_ORDER));
   temp_item->status |= REC_ITEM_ONL_IMMEDIATELY;
  }
 }
 /*
 ** If the global field still has the value of the passed in
 ** rec_order_num, it means that there is no more item with
 ** strict recovery order in the list. Set the golbal field to 0.
 */
 if(*first_strict_order == rec_order_num )
 {
  *first_strict_order = 0;
 }
 return;
}
/*
** REC_BUILD_RECOVERY_INFO
**
**
** Purpose:
** This function determines the order in which databases will be
** recovered and builds a list of databases to recover in the
** REC_ORDER_INFO structure in Resource.
**   First, it reads sysattributes in the meta data database to determine
** any user-specified recovery order.
** Next, any database whose recovery order has not been specified, is
** added to the list in dbid order.
**
** NOTE that the databases which were in load (DBT_INLDDB) or marked
** not to be recovered (DBT_USE_NOTREC) will be skipped with
** appropriate messages printed.
**
** Parameters:
**  meta_dbt - At boot time this is the master database;
**     At failover time this is the master_companion
**     database.
**       We will read sysattributes and sysdatabases from
**     this passed in meta database.
** backout_ctx - The backout context structure initialized by caller.
**     This function will fill some fields in this structure.
**
** Returns:
** num_dbs_to_recover >= 0 Number of databases that will be recovered
**    < 0  Failure
**
**
** Side Effects:
** A private chunk of memory might be allocated and hung off Resource.
** The list of databases to be recovered will be stored in this chunk
** of memory in certain order.
**
**  1. allocate memory that is big enough to hold structures for all the
** offline dbs.
**  NOTE: we may not necessarily fill all of the items, for some of
**  the offline dbs may have some status bits set which will result
**  in the dbs not being recovered.
**
**  2. rec_order_size will be set at the end of this function
**  to the number of db items that are actually stored in the structure.
**
**  NOTE: This probably wouldn't be the precise number of databases that
**  will be recovered (which is represented by num_dbs_to_recover).
** The reason is:
**  All the databases with special recovery order are stored in the
** rec_order_info structure. If later on we find that any such database
** has status bits that require the dbs not to be recovered, we will NOT
** remove the item from the structure.
**  Instead, we reset the REC_ITEM_NOT_RECOVERED bit and set the
**  REC_ITEM_SKIP_RECOVERY in the item. We also decrement the
** num_dbs_to_recover value. In this way, rec_getnextdb_to_recover()
** will be able to skip this item.
```

```
**
** Synchronization:
** No need to use spinlock protection when we access the fields in
** Resource, as we know that even with parallel recovery, this function
** is called by the tuning thread before any recovery thread is spawned.
** So there is no concurrent access of these fields at this time.
** This is also true for ha failover recovery, where the tuning thread
** is the thread that is executing the failover.
**
** History
** written 07/30/97(raghu)
** 4/29/02 (fzhou) - add to the recovery list the databases which will
**     be recovered in default order (i.e. dbid).
*/
int
rec_build_recovery_info(REC_CALLER_BKOUT_CTX *backout_ctx, DBTABLE *meta_dbt)
{
 ATTRINFO       attrinfo;
 int   attr_return;
 int  rec_order_num;
 int  rec_info_memsize;
 int  cnt;
 int  *first_strict_order;
 int  num_offline_dbs;    /* the number of databases that
         ** need recovery. this is the
         ** number stored in Resource.
         */
 int  special_order_dbs;  /* the number of databases that have
         ** recovery order specified in
         ** sysattributes.
         */
 dbid_t  metadbid;
 dbid_t  curdbid;
 SYB_BOOLEAN boot_recovery;
 RECOVERY_INFO *recovery_info;
 REC_ORDER_INFO *rec_order_info;
 REC_ORDER_ITEM *rec_order_item;
 SDES  *db_sdes; /* sdes for sysdatabases */
 BUF  *dbbuf;
 DATABASE *sysdb;  /* store the sysdatabases row. */
 BYTE  *stat3p;
 int  stat3len; /* store the length field
    ** for status3 in the
    ** collocate() call.
    */
 int32  datstat3;
 int  dbnlen;  /* store the length for dbname.*/
 BYTE  *namep;
 char  dbname[MAXNAME];
 SARG  dbkeys[6];
```

```
short           datstat;
SYB_BOOLEAN strict_order_found;
/* Initialize */
strict_order_found = FALSE;
metadbid = meta_dbt->dbt_dbid;
/* Boolean used to figure out if we are in boot time recovery
** or in mount time recovery.
*/
boot_recovery = (metadbid == MASTERDBID);
/* Localize the RECOVERY_INFO in Resource. */
recovery_info = &Resource->rrecovery_info;
/*
** Obtain the number of databases that will need recovery
** from Resource.
** This number will be used to allocate memory for rec_order_info.
**
** 1. For boot time recovery, this number is the number of
**    offline databases + the number of failed over databases.
**    This is because after master db is recovered, when we count
**    the number of offline databases in rec__set_numdb(), if the
**    database is DBT3_FAILEDOVER_DATABASE, it will only not be
**    accounted for in rnum_offlinedb, but only in cnum_offlinedb.
*/
if(boot_recovery)
{
 num_offline_dbs = Resource->rnum_offlinedb +
    Resource->companion_info.cnum_offlinedb;
}
else
{
 /*
 ** 2. for failover recovery, this is the number of databases
 **    that got mounted, plus the number of user created
 **    tempdbs. User created tempdbs were not mounted, but they
 **    are still in the system catalogs (i.e. sysattributes
 **    if they are special recovery order, and sysdatabases) in
 **    master_companion, and therefore they will be visible
 **    for our scan of these two catalogs later. Allocate
 **    space enough to hold all dbs.
 */
 num_offline_dbs = Resource->companion_info.cnum_offlinedb +
    Resource->companion_info.cnum_tempdbs;
}
/*
** If there is no offline user dbs to recover, do not need to
** populate the rec_order_info structure. Just return to caller.
*/
if (num_offline_dbs == 0)
{
 return 0;
}
/*
** Compute memory size for the rec_order_info which will hold all
** rec_order_items.
**
** Note:
** we need to allocate one more REC_ORDER_ITEM than the number of
** offline databases because:
**
** -- the rec_order_item field in REC_ORDER_INFO will point to
** the start of an array of rec_order_items, each of which is
** specific for one offline database.
** -- The first entry of the array will _NOT_ be used because the
** special recovery order id that the user can specify (which is
** stored in sysattributes) starts at 1 and it is easier
** to keep the index of the array and the item order id in sync.
*/
backout_ctx->rec_info_memsize = rec_info_memsize =
  (num_offline_dbs + 1) * REC_ORDER_ITEM_SIZE
     + REC_ORDER_INFO_SIZE;
/* Now get the memory. */
backout_ctx->rec_order_info = rec_order_info =
 (REC_ORDER_INFO *) ALLOC_PRIVATE_MEMCHUNK(rec_info_memsize);
/*
** TESTING POINT: when trace flag 3462 is on, force the allocation
** to fail.
*/
if (TRACECMDLINE(RECOVER, 62) && rec_order_info != NULL)
{
 backout_ctx->rec_order_info = rec_order_info = NULL;
 FREE_PRIVATE_MEMCHUNK(backout_ctx->rec_order_info,
     backout_ctx->rec_info_memsize);
}
if (rec_order_info == NULL)
{
 mnt_ex_print(EX_NUMBER(RECOVER2, REC_MEMALLOC_FAIL), EX_INTOK, 1);
 goto error_return;
}
/* Initialize the chunk of memory */
MEMZERO(rec_order_info, rec_info_memsize);
/* Make the block of memory available globally off Resource. */
recovery_info->rec_order_info = rec_order_info;
/* save rec_info_memsize to be used during FREE_PRIVATE_MEMCHUNK */
rec_order_info->rec_info_memsize = rec_info_memsize;
/* Initialize the rec_order_size to 0 */
rec_order_info->rec_order_size = 0;
/*
** The start of the array of rec_order_item will be after all the
** header fields in rec_order_info.
*/
```

```
       rec_order_item = rec_order_info->rec_order_item =
  (REC_ORDER_ITEM *)( ((ptrdiff_t)rec_order_info) +
       ((ptrdiff_t)REC_ORDER_INFO_SIZE) );
/* Get the local pointer to the global field */
first_strict_order = &rec_order_info->first_offline_strict_order_item;
/* 1. read the rows from sysattributes in the meta db. */
rec_order_num = 1;
do
{
 REC_ORDER_INIT_STRUCT(attrinfo);
 attrinfo.aiintvalue = rec_order_num;
  attr_return = attrib_getrow(&attrinfo, meta_dbt);
 if (attr_return == ATTR_ROW_FOUND)
 {
  rec_order_item[rec_order_num].dbid = attrinfo.aiobject;
  /*
  ** Get the mode by which the database will be brought
  ** online.
  ** If it has "strict" stored in sysattributes,
  ** this database must be brought online in a STRICT
  ** mode. I.e. all databases before this database need to
  ** be onlined before this database can be onlined,
  ** and all databases after this database can NOT be
  ** onlined until this database is onlined.
  */
  if (((int) attrinfo.aicommlen != 0) &&
      (strncmp((char *)attrinfo.aicomments, "strict",
        (int) attrinfo.aicommlen)) == 0)
  {
   rec_order_item[rec_order_num].status |=
     REC_ITEM_ONL_WITH_STRICT_ORDER;
   /*
   ** If this is the first item with strict online
   ** order, set it in rec_order_info.
   */
   if (strict_order_found == FALSE)
   {
    *first_strict_order = rec_order_num;
    strict_order_found = TRUE;
   }
  }
  else if (strict_order_found == FALSE)
  {
   /*
   ** This database doesn't have
   ** "strict" set in sysattributes and none of
   ** the databases before it has STRICT mode,
   ** this database will be brought  online in a
   ** RELAX mode.
   ** If this database does not have a strict
   ** order but databases before it has strict
   ** order, then this cannot be marked RELAX
   ** as databases before this have to online
   ** before this does
   */
   rec_order_item[rec_order_num].status |=
     REC_ITEM_ONL_IMMEDIATELY;
  }
  rec_order_item[rec_order_num].status |=
     REC_ITEM_NOT_RECOVERED;
  rec_order_num++;
 }
} while (attr_return == ATTR_ROW_FOUND);
/*
** TESTING POINT: If trace flag 3463 is on, force the read
** of special recovery order to fail.
*/
if (TRACECMDLINE(RECOVER, 63))
{
 attr_return = ATTR_ERROR;
}
/*
** Read from sysattributes failed, print an error.
*/
if (attr_return == ATTR_ERROR)
{
 ex_logprint(EX_NUMBER(RECOVER2, REC_ORDER_READFAIL), EX_INFO, 1);
 goto error_return;
}
/* save the number of user specified recovery orders */
special_order_dbs = rec_order_num - 1;
if (TRACECMDLINE(RECOVER, 56))
{
 scerrlog("There are %d offline user databases, among which %d have special recovery order.\n",
   num_offline_dbs, special_order_dbs);
}
/*
** Set the number of databases to recover to the number of databases
** with special recovery order first. This counter will be changed
** as we read in more databases without special recovery order, if
** there is any.
*/
rec_order_info->num_dbs_to_recover = special_order_dbs;
/*
** 2. Read from sysdatabases for other databases. Put the ones
** that don't have specified recovery order to the end of the list
** in the order of their dbid.
*/
if (boot_recovery)
{
 db_sdes = OPEN_SYSTEM_TABLE((objid_t) SYSDATABASES, MASTERDBID,
     Resource->rmasterdbt);
}
else
{
 db_sdes = OPEN_USER_TABLE((objid_t) SYSDATABASES, metadbid, 1,
   (BYTE *)"sysdatabases",
   (sizeof("sysdatabases") - 1));
}
if (!db_sdes)
 goto error_return;
/* remember the sdes in backout structure. */
backout_ctx->db_sdes = db_sdes;
/*
** TESTING POINT: If trace flag 3464 is on, simulate a fatal error
** while opening sysdatabases. We have to raise the error here
** after the sdes has been remembered in the backout_ctx structure,
** otherwise, it won't get cleaned up.
*/
if (TRACECMDLINE(RECOVER, 64))
{
 ex_raise(RECOVER, REC_RETURN, EX_CMDFATAL, 1);
}
db_sdes->sstat |= (SS_FGLOCK | SS_L1LOCK);
curdbid = MODELDBID;
/*
** Initialize search arg array based on whether we are
** in boot or mount time recovery.
*/
initarg(db_sdes, dbkeys, (boot_recovery) ? ARRAY_LEN(dbkeys) : 2);
/* These two are common for both boot and mount time recovery */
setarg(db_sdes, &Sysdatabases[DAT_DBID], GT,
 (BYTE *) &curdbid, sizeof (dbid_t));
setarg(db_sdes, &Sysdatabases[DAT_NAME], NE,
      Sybsystemdb, sizeof (Sybsystemdb) -1);
/* Boot time recovery wants to filter out some more databases */
if (boot_recovery)
{
 /*
 ** Skip following system databases during boot
 ** recovery as they were recovered separately earlier.
 */
 setarg(db_sdes, &Sysdatabases[DAT_NAME], NE, Procdb,
  (sizeof(Procdb) - 1));
 setarg(db_sdes, &Sysdatabases[DAT_NAME], NE, Secdb,
  (sizeof (Secdb) - 1));
 setarg(db_sdes, &Sysdatabases[DAT_NAME], NE,
       Master_Companion, (sizeof(Master_Companion) -1));
 setarg(db_sdes, &Sysdatabases[DAT_NAME], NE,
       Sybsystemdb_Companion,
        (sizeof(Sybsystemdb_Companion) - 1));
}
startscan(db_sdes, NC1_DATABASE, SCAN_NORMAL);
/*
** rec_order_num is again used here as the index to the array of
** rec_order_item. We reinitialize it to the correct index number.
*/
rec_order_num = special_order_dbs + 1;
while ((dbbuf = getnext(db_sdes)) != NULL)
{
 sysdb = (DATABASE *) db_sdes->srow;
 curdbid = GETSHORT(&sysdb->datdbid);
 datstat = GETSHORT(&sysdb->datstat);
 stat3p = collocate((BYTE *) sysdb,
                         Sysdatabases[DAT_STATUS3].scoloffset,
                         IND_GET_01ROW_MINLEN(db_sdes->sdesp),
                         sizeof(sysdb->datstat3),
                         SDES_ROWFORMAT(db_sdes), &stat3len);
 datstat3 = (stat3p != (BYTE *) NULL) ?
    GETLONG(stat3p) : (int32) 0;
 /* get dbname for message printing. */
 namep = collocate((BYTE *) sysdb,
   Sysdatabases[DAT_NAME].scoloffset,
   OFFSETOF(DATABASE, datlen),
   Sysdatabases[DAT_NAME].scollen,
   SDES_ROWFORMAT(db_sdes), &dbnlen);
 MEMMOVE(namep, dbname, dbnlen);
 /*
 ** Check to see if this database has a special
 ** recovery order defined, i.e. curdbid is already in
 ** the list.
 */
 for (cnt = 1; cnt <= special_order_dbs; cnt++)
 {
  if (rec_order_item[cnt].dbid == curdbid)
   break;
 }
 /*
 ** For database that will skip recovery, if it has a
 ** special recovery order, we need to reset the
 ** REC_ITEM_NOT_RECOVERED bit so that it won't be picked up
 ** for recovery later.
 ** Also mark the item as REC_ITEM_SKIP_RECOVERY and
 ** decrement the counter of num_dbs_to_recover.
 **
 ** Macro DB_SKIP_RECOVERY decides if the passed in database
 ** will skip recovery.
 ** The following databases will not be recovered:
 ** 1. database is in load OR
 ** 2. database is marked bypass OR
** 3. database is user created tempdb AND server is
**    doing failover recovery.
*/
if (DB_SKIP_RECOVERY(datstat, datstat3) &&
   (cnt <= special_order_dbs))
{
 /*
 ** If the database had strict online order
 ** specified, print a warning stating that
 ** this order will not be maintained.
 */
 if (rec_order_item[cnt].status &
   REC_ITEM_ONL_WITH_STRICT_ORDER)
 {
  if (datstat & DBT_USE_NOTREC)
  {
   mnt_ex_print(EX_NUMBER(RECOVER2, REC_BYPASS_NOSTRICT), EX_INFO, 1,
    dbnlen, dbname);
  }
  else if (datstat & DBT_INLDDB)
  {
   mnt_ex_print(EX_NUMBER(RECOVER2, REC_LOAD_NOSTRICT), EX_INFO, 1,
    dbnlen, dbname);
  }
  rec_order_item[cnt].status &=
   ~REC_ITEM_ONL_WITH_STRICT_ORDER;
  /*
  ** If this item is the first offline
  ** item with strict order, need to
  ** find the next strict item, and also
  ** set the REC_ITEM_ONL_IMMEDIATELY bit
  ** in all items in between.
  **
  ** We pass in the (rec_order_num - 1) as
  ** the end of the search to make sure that
  ** the function also visits all the items
  ** that are already put in the
  ** rec_order_item list, including those
  ** without special recovery order,
  */
  if (cnt == *first_strict_order)
  {
   (void) rec__set_next_strict_order_item(
    cnt, (rec_order_num - 1),
    FALSE);
   /* If there is no other
   ** strict item, set the
   ** strict_order_found to FALSE.
   */
   if (*first_strict_order == 0)
   {
    strict_order_found = FALSE;
   }
  }
 }
 /*
 ** Since this database will not be recovered, clear
 ** all status bits regarding recovery state in the
 ** recovery item, and set the REC_ITEM_SKIP_RECOVERY
 ** indicating the database will skip recovery.
 ** Also modify the book keeping field in rec_order_info.
 */
 rec_order_item[cnt].status &=
    ~(REC_ITEM_RECOVERY_STATE);
 rec_order_item[cnt].status |=
    REC_ITEM_SKIP_RECOVERY;
 rec_order_info->num_dbs_to_recover --;
}
/*
** Check for incomplete load database command. If so, skip
** this database.
*/
if ((datstat & DBT_INLDDB) && !(datstat & DBT_USE_NOTREC))
{
 ex_callprint(EX_NUMBER(OPENDBM, DB_LOAD), EX_INFO, 1,
       curdbid);
 ex_callprint(EX_NUMBER(RECOVER2, REC_CONTINUE_NEXTDB),
   EX_INFO, 1);
 continue;
}
/*
** check for special mode where we are not
** going to recover database regardless... this is an emergency
** feature to allow access to dbs that are in trouble
*/
if (datstat & DBT_USE_NOTREC)
{
 ucierrlog(NOFAC_SERVER, UTILS_BYPSSRECOVDBID, curdbid);
 make_log_consistent(curdbid);
 continue;
}
/*
** If we have found the curdbid in the recovery items list.
** Continue with the next db.
*/
if (cnt <= special_order_dbs)
{
 /*
 ** If this is a user created tempdb, set the
 ** status in the item before continue to next db.
 ** For failover recovery, the database will not
 ** be recovered, because REC_ITEM_SKIP_RECOVERY
 ** was already set earlier in the item. However,
 ** we will still set the REC_ITEM_USER_TEMPDB
 ** in the item to indicate the type of the database
 ** represented by the item.
 */
 if (datstat3 & DBT3_USER_TEMPDB)
 {
  rec_order_item[cnt].status |=
    REC_ITEM_USER_TEMPDB;
 }
 continue;
}
/*
** curdbid is the next database to be recovered. Save it in
** the list.
*/
rec_order_item[rec_order_num].dbid = curdbid;
/*
** If this is a user createdb tempdb, set the status
** in the item indicating so.
*/
if (datstat3 & DBT3_USER_TEMPDB)
{
 rec_order_item[rec_order_num].status |=
   REC_ITEM_USER_TEMPDB;
 /*
 ** Failover recovery does not recover user created
 ** tempdbs, so mark it as skipped recovery, and
 ** continue to the next database. Since
 ** REC_ITEM_NOT_RECOVERED bit will not be set in
 ** the item, and num_dbs_to_recover is not
 ** incremented for this item, the database will
 ** be skipped later by rec_getnextdb_to_recover().
 */
 if (IS_FAILOVER_THREAD())
 {
  rec_order_item[rec_order_num].status |=
     REC_ITEM_SKIP_RECOVERY;
  rec_order_num++;
  continue;
 }
}
/*
** Increment the counter for number of databases to recover.
*/
rec_order_info->num_dbs_to_recover ++;
/*
** Determine the mode by which the database will be
```

```
 ** brought online. Since this database doesn't have a
 ** special recovery order defined, it can't have a strict
 ** online order set. If there is no item before this
 ** db that has strict online order, this database will have
 ** a relaxed online order.
 */
 if (strict_order_found == FALSE)
 {
  rec_order_item[rec_order_num].status |=
     REC_ITEM_ONL_IMMEDIATELY;
 }
 rec_order_item[rec_order_num].status |= REC_ITEM_NOT_RECOVERED;
 rec_order_num++;
}
/*
** Set the rec_order_size to the actual number of database
** items that are stored in the rec_order_info structure.
*/
rec_order_info->rec_order_size = rec_order_num - 1;
/* End the scan of sysdatabases and close the table. */
endscan(db_sdes);
CLOSE_SDES(&backout_ctx->db_sdes);
/*
** Set the status indicating the start of parallel recovery only
** when we found any database to recover.
*/
if (rec_order_info->num_dbs_to_recover > 0)
 recovery_info->status |= REC_INFO_PARALLEL;
return (rec_order_info->num_dbs_to_recover);
error_return:
/* If we have opened sysdatabases, close it. */
CLOSE_SDES(&backout_ctx->db_sdes);
/*
** If we have allocated memory for rec_order_info in resource,
** we need to release that memory.
*/
if (backout_ctx->rec_order_info != (REC_ORDER_INFO *)NULL)
{
 backout_ctx->rec_order_info =
  Resource->rrecovery_info.rec_order_info =
     (REC_ORDER_INFO *) NULL;
 FREE_PRIVATE_MEMCHUNK(rec_order_info,
     backout_ctx->rec_info_memsize);
}
/*
** Set the bit in rrecovery_info.status indicating that server
** will recovery databases in dbid order.
*/
recovery_info->status |= REC_INFO_BY_DBID;
/*
```

```
 ** The read of offline databases into Resource->rrecovery_info
 ** failed. We will recover the databases in serial by the order
 ** of dbid.
 */
 mnt_ex_print(EX_NUMBER(RECOVER2, REC_READ_ORDER_FAIL), EX_INFO, 1);
 return (-1);
}
/*
** REC_RUN_PARALLEL_RECOVERY
**
**
** Purpose:
** Tune and spawn recovery threads. Spawn more recovery threads only
** when the sampled statistics shows that the I/O subsystem is able to
** handle the request of parallel I/O. The maximum number of recovery
** threads to spawn is determined by four factors:
** 1. the configuration value of "max concurrently recovered db";
** 2. the number of engines online;
** 3. the configuration value of "number of open databases";
** 4. the number of databases that need to be recovered.
**
** After reaching a stable state (i.e. no more spawning of recovery
** threads), this initial thread will sleep on REC_INFO_PARALLEL status
** bit until it is cleared. This status bit is cleared by the last
** recovery thread before it exits.
**
** Parameters:
** backout_ctx - Backout context structure initialized by caller.
** rec_caller_arg - Structure contains the information needed to recover
**   user created tempdbs, and the field to pass to the
**   spawned recovery threads.
**
** Returns:
** nothing.
**
** Synchronization:
** Use spinlock protection when access recovery_info in Resource,
** because while this is running, other recovery threads could be
** accessing this field.
**
** History:
** (5 '02 fzhou) - created
*/
void
rec_run_parallel_recovery(REC_CALLER_BKOUT_CTX *backout_ctx,
   REC_CALLER_ARG *rec_caller_arg)
{
 LOCALPSS(pss);
 int  config_value;
 int32  num_engines_to_use;
```

```
int32  num_dbt_descriptors;
int  num_dbs_to_recover;
int  max_recovery_parallelism;
int  max_num_threads_to_spawn;
int  num_spawned;
int  optimal_num_threads;
int  prev_stat;  /* the previous statistics
     ** collected.
     */
int  statistics_results;
int  sleepstat;
int  stat_index;
SYB_BOOLEAN failover_recovery;
kpid_t  kpid;
int  sample_average;
/* local pointer to the global structure.  */
RECOVERY_INFO *recovery_info;
REC_ORDER_INFO *rec_order_info;
REC_ORDER_ITEM *cur_rec_item;
/* Initialize local variables. */
config_value = max_recovery_parallelism = 0;
num_dbt_descriptors = num_engines_to_use = num_dbs_to_recover = 0;
max_num_threads_to_spawn = num_spawned = optimal_num_threads = 0;
prev_stat = 0;
sample_average = 0;
failover_recovery = IS_FAILOVER_THREAD();
recovery_info = &Resource->rrecovery_info;
rec_order_info = recovery_info->rec_order_info;
if (TRACECMDLINE(RECOVER, 56))
{
 TRACEPRINT("The global structure after we read into it: \n");
 prRECOVERY_INFO(recovery_info);
}
/*
** If trace flag 3474 is not on, tune the buffer pools in the
** default data cache for recovery:
** 1. tune the sizes of the pool with largest mass and the default
**    pool by redistributing buffers between the pools.
**    (If the largest mass pool doesn't exist, it will be created.)
** 2. apf for the largest mass pool and the default pool.
*/
if (!TRACECMDLINE(RECOVER, 74))
{
 rec__tune_bufpools(REC_SET_CONFIG, DEFAULT_CACHE_ID);
}
/*
** The max number of recovery threads to spawn is determined by
** the following factors:
** 1. the maximum degree of parallelism.
** 2. the number of databases to recover.
```

```
**
** The maximum degree of parallelism is determined by the user
** specified config value of "max concurrently recovered db" and
** other server resource limits.
**
** The limits are:
** 1. the number of active engines:
**    the number of engines that will be used by recovery is
**    (number of active engines - 1),
** 2. the configured number of open databases:
**    the max number of dbtable descriptors that can be used by recovery
**    is (the configured value - 3), because master, system tempdb,
**    and model db would have already used 3 dbtable descriptors.
** The max degree of parallelism for recovery cannot exceed either
** of the limits.
**
** If the configured value is not DEFAULT, before we use the
** config value, we have to verify the value against those limits.
** Otherwise, (the configured value is 0 (DEFAULT value)), we will
** determine the config value using those limits.
**
** Even if these limits are enforced when the config parameter
** is verified, it is possible for them to change before recovery
** happens. And therefore they have to be checked again.
**
** If the value of the configuration parameter fails the test, the
** max degree of parallelism will be set to the smaller value of
** the two limits. Otherwise, the max degree of parallelism will be
** set to the value of the configuration parameter.
**
** If the configured value for "max concurrently recovered db" is
** DEFAULT, we will set max degree of parallelism to the smaller
** value of the two limits.
**
** In the sp_configure procedure, it is ensured that during recovery,
** the value of "number of open databases" cannot be reduced.
*/
config_value = CFG_GETCURVAL(cmaxconcurrentrecdb);
num_engines_to_use = Kernel->kenumonline - 1;
num_dbt_descriptors = CFG_GETCURVAL(cdbnum) - 3;
/* If there is only 1 active engine, do serial recovery. */
if (num_engines_to_use == 0)
{
 max_recovery_parallelism = 1;
}
else
{
 /*
 ** Calculate the maximum degree of parallelism based
 ** on number of active engines and number of open
 ** databases.
 **
 ** Even when there are two active engines (where
 ** max_recovery_parallelism will be 1), we will do serial
 ** recovery without spawning any recovery thread, because
 ** there is no concurrent recovery with 1 recovery thread.
 */
 max_recovery_parallelism = (num_engines_to_use >
     num_dbt_descriptors) ?
         num_dbt_descriptors :
         num_engines_to_use;
}
/*
** If user has specified a value for the configuration parameter,
** AND it exceeds the limit, print a warning and use the
** max_recovery_parallelism instead of the config_value for recovery.
**
** If trace flag 3458 is on, always use the config value as the
** max_recovery_parallelism.
*/
if (config_value != 0)
{
 if (!(TRACECMDLINE(RECOVER, 58)) &&
  (config_value > max_recovery_parallelism))
 {
  mnt_ex_print(EX_NUMBER(RECOVER2, REC_CONFIG_TOO_BIG), EX_INFO, 1,
   config_value,
   max_recovery_parallelism,
   max_recovery_parallelism);
 }
 else
 {
  /*
  ** If config value doesn't exceed the limit, OR
  ** trace flag 3458 is on, use the config_value as
  ** the max_recovery_parallelism.
  */
  max_recovery_parallelism = config_value;
 }
}
num_dbs_to_recover = rec_order_info->num_dbs_to_recover;
/*
** The max number of recovery threads that we will spawn is a
** function of the max_recovery_parallelism and number of dbs to
** recover.
*/
max_num_threads_to_spawn =
  (max_recovery_parallelism < num_dbs_to_recover) ?
   (max_recovery_parallelism) :
   (num_dbs_to_recover);
```

```
if (TRACECMDLINE(RECOVER, 56))
{
 scerrlog("the maximum number of recovery processes that will be used for recovery is %d.\n",
     max_num_threads_to_spawn);
}
/*
** If the maximum recovery parallelism is 1, we will do serial
** recovery. There is no need to spawn any recovery thread.
** 1. Clear the REC_INFO_PARALLEL bit,
** 2. Clear the field in rec_order_info which keeps track of the
**    items with strict recovery item. That field is used to
**    maintain strict recovery order during parallel recovery, so
**    it is not needed during serial recovery.
*/
if (max_num_threads_to_spawn == 1)
{
 recovery_info->status &= ~(REC_INFO_PARALLEL);
 rec_order_info->first_offline_strict_order_item = 0;
}
/* Initialize the number of running recovery threads. */
rec_order_info->num_rec_threads =
 rec_order_info->optimal_num_rec_threads = 0;
/*
** Only spawn recovery threads if we are still in parallel recovery.
*/
while ((num_spawned < max_num_threads_to_spawn) &&
 (recovery_info->status & REC_INFO_PARALLEL))
{
 /*
 ** TESTING POINT: If trace flag 3465 is on, raise a
 ** non-fatal error after 1 recovery thread has been spawned.
 */
 if (TRACECMDLINE(RECOVER, 65) && (num_spawned == 1))
 {
  ex_raise(RECOVER, REC_RETURN, EX_RESOURCE, 5);
 }
 /*
 ** Store the engine number in the context structure which
 ** will be passed to the handler so that the spawned thread
 ** can be affiliated with the specific engine under trace
 ** flag 3455.
 ** NOTE:  We can do this because we know that the number of
 ** spawned recovery threads must less than or equal to the
 ** number of engines.
 */
 rec_caller_arg->engine_num = (engid_t) num_spawned;
 kpid = spawn_handler(rec__parallel_hdlr, MIDUPRI,
    (FNIPARAM) rec_caller_arg, 0,
    rec_thread_cleanup, NULL,
    SPAWN_SYSTEM_TASK);
```

```
/*
** TESTING POINT: If trace flag 3469 is on, simulate an error
** return from spawn_handler(), after 1 recovery thread has
** been spawned,
** The spawned thread has been asynchronously terminated in
** in rec__parallel_hdlr()
*/
if (TRACECMDLINE(RECOVER, 69) && (num_spawned == 1) &&
  (kpid >= 0))
{
 scerrlog("TESTING: Simulating errors while spawning a recovery thread.\n");
 kpid = -1;
}
if (kpid < 0)
{
 /*
 ** Even though we couldn't spawn any more recovery
 ** threads, we will not fail the recovery.
 ** Recovery will use the currently available threads.
 ** Even if there is no recovery thread, server will
 ** recover databases in serial mode.
 */
 mnt_ex_print(EX_NUMBER(RECOVER2, REC_TASK_SPAWN_FAIL), EX_INTOK, 1);
 /* Stop spawning any more threads */
 break;
}
/* Successfully spawned one recovery thread */
else
{
 /*
 ** Increment num_spawned to indiate we have
 ** successfully spawned a recovery thread.
 */
 num_spawned ++;
 /*
 ** Increment the current number of running recovery
 ** threads in rec_order_info.
 ** Don't do a direct assignment from num_spawned
 ** to num_rec_threads, because there could be
 ** recovery threads that were spawned but have
 ** exited and thus have decremented the same field.
 **
 ** Set the optimal number of rec threads to the
 ** current running number of threads.
 **
 ** This has to be done under spinlock, for the
 ** recovery threads have already started, which
 ** could access this field.
 */
 P_SPINLOCK(Resource->ha_spin);
```

```
rec_order_info->num_rec_threads++;
rec_order_info->optimal_num_rec_threads =
  rec_order_info->num_rec_threads;
rec_order_info->latest_spawned_thread_kpid = kpid;
V_SPINLOCK(Resource->ha_spin);
/* start the process */
upstart(kpid);
/*
** Don't do the statistics sampling if 3458 is
** turned on, just spawn as many threads as
** the config parameter indicates
*/
if (TRACECMDLINE(RECOVER, 58))
{
 continue;
}
/*
** Collect statistics for this thread and determine
** if we have found an unacceptable degradation of
** statistics results over the previous statistics.
*/
statistics_results =
 rec__stat_degraded(num_spawned, &prev_stat);
switch (statistics_results)
{
 /*
 ** If recovery has already completed, do not
 ** spawn any more thread.
 */
 case REC_RECOVERY_COMPLETE:
  break;
 /*
 ** If the new statistics was invalid or it
 ** didn't show any unacceptable degradation
 ** over the previous statistics, continue
 ** to spawn the next recovery thread.
 */
 case REC_INVALID_STAT:
 case REC_STAT_NO_DEGRADATION:
  continue;
 /*
 ** If there was an unacceptable degradation,
 ** take either of the following path:
 ** 1. If user has specified a value for
 **    the configuration parameter, continue
 **    to spawn recovery threads in spite of
 **    what server finds. But keep a record
 **    of what server thinks is the optimal
 **    number of recovery threads, which will
 **    be printed later as an advisory to
```

```
   **    the user.
   ** 2. If user uses DEFAULT for the config
   **    parameter, decrement the optimal number
   **    of recovery threads by one because server
   **    was found to be good to handle all the
   **    threads until we spawned the last one.
   **    Stop spawning any more thread, and later
   **    we will freeze one thread.
   */
   case REC_STAT_DEGRADATION:
    P_SPINLOCK(Resource->ha_spin);
    if ((config_value != 0) &&
     (optimal_num_threads == 0))
    {
     /*
     ** First remember the current
     ** number of running running
     ** threads under spinlock
     ** protection.
     */
     optimal_num_threads =
      rec_order_info->num_rec_threads;
     V_SPINLOCK(Resource->ha_spin);
     /*
     ** Decrement the value
     ** because server found the
     ** optimal number of threads
     ** is one less than the
     ** currently online threads.
     */
     optimal_num_threads --;
     continue;
    }
    else
    {
     rec_order_info->optimal_num_rec_threads --;
     V_SPINLOCK(Resource->ha_spin);
     break;
    }
   default:
    /*
    ** should have no other return values.
    */
    SYB_ASSERT(0);
    break;
  }
  /* break out of the while loop. */
  break;
 }
} /* END of the while loop to spawn threads*/
```

```
if (rec_order_info->optimal_num_rec_threads != 0)
{
 /*
 ** If user has specified a value for the configuration
 ** parameter, server will not stop spawning threads even if it
 ** finds performace degradation. Since we have kept a record
 ** of what server found to be the optimal number of recovery
 ** threads, print it here as an advisory if it is not the same
 ** as the number of threads spawned.
 */
 if ((config_value) && (optimal_num_threads != 0) &&
  (optimal_num_threads != max_num_threads_to_spawn))
 {
  mnt_ex_print(EX_NUMBER(RECOVER2, REC_NUM_PROCESSES_ON_REQUEST), EX_INFO, 1,
    optimal_num_threads,
    max_num_threads_to_spawn);
 }
 else
 {
  mnt_ex_print(EX_NUMBER(RECOVER2, REC_NUM_PROCESSES), EX_INFO, 1,
   rec_order_info->optimal_num_rec_threads);
 }
}
else
{
 mnt_ex_print(EX_NUMBER(RECOVER2, REC_INFO_SERIAL_RECOVERY), EX_INFO, 1);
}
/* Freeze a recovery thread if needed. */
rec__freeze_recovery_thread();
/* Set the bit indicating that tuning has completed. */
P_SPINLOCK(Resource->ha_spin);
recovery_info->status |= REC_INFO_TUNE_COMPLETE;
V_SPINLOCK(Resource->ha_spin);
/*
** If trace flag 3456 is on, print the sampling statistics to errorlog.
*/
if (TRACECMDLINE(RECOVER, 56))
{
 for (stat_index = 1; stat_index <= num_spawned; stat_index++)
 {
  cur_rec_item =
   &rec_order_info->rec_order_item[stat_index];
  sample_average = (cur_rec_item->sample_result[0] +
    cur_rec_item->sample_result[1] +
    cur_rec_item->sample_result[2])/3;
  /*
  ** Print out the sample values for each statistics
  ** that server has collected.
  */
  mnt_ex_print(EX_NUMBER(RECOVER2, REC_SAMPLE_VALUES), EX_INFO, 1,
    stat_index,
    cur_rec_item->sample_result[0],
    cur_rec_item->sample_result[1],
    cur_rec_item->sample_result[2],
    sample_average);
 }
 prRECOVERY_INFO(recovery_info);
}
P_SPINLOCK(Resource->ha_spin);
/* If there is still a recovery thread running ... */
if (rec_order_info->num_rec_threads > 0)
{
 /*
 ** ... this initial thread will go to sleep on the
 ** REC_INFO_PARALLEL bit until this bit is reset,
 ** which will be done by any of the recovery thread(s) when all
 ** the dbs to be recovered have been recovered.
 **
 ** The "wake on attention" parameter is FALSE because
 ** the recovery process cannot be killed by user's
 ** attention.
 **
 ** Since this is not in a performance sensitive code path,
 ** no need to add a mda unique call identifier for this sleep.
 */
 while (recovery_info->status & REC_INFO_PARALLEL)
 {
  V_SPINLOCK(Resource->ha_spin);
  sleepstat = upsleepgeneric(SYB_EVENT_NON_STRUCT(&recovery_info->status),
   (char *)&recovery_info->status,
   sizeof(recovery_info->status),
   REC_INFO_PARALLEL, FALSE, UNUSED);
  P_SPINLOCK(Resource->ha_spin);
 }
}
/*
** No need to spinlock any more, because there is no recovery threads
** running. We are the only one accessing this global structure.
*/
V_SPINLOCK(Resource->ha_spin);
/*
** If failover has failed, do not try to recover more
** databases. Return to caller.
*/
if ((failover_recovery) &&
 (recovery_info->status & REC_INFO_FAILOVER_FAIL))
{
 goto exit_point;
}
/*
```

```
** If we were woken up because of the user instruction to go back
** to serial recovery (by setting the config parameter to 1),
** clear the bit because now we are back to serial recovery.
*/
if (recovery_info->status & REC_INFO_GOBACK_TO_SERIAL)
{
 mnt_ex_print(EX_NUMBER(RECOVER2, REC_USER_SERIAL_RECOVERY), EX_INFO, 1,
    (rec_order_info->num_dbs_to_recover -
     rec_order_info->num_dbs_done));
 recovery_info->status &= ~(REC_INFO_GOBACK_TO_SERIAL);
}
/*
** If we still have databases to recover, recover them serially.
*/
if (rec_order_info->num_dbs_done != rec_order_info->num_dbs_to_recover)
{
 if (failover_recovery)
 {
  if (rec_failover_recover_dbs(backout_ctx, SERIAL)
        == FAIL)
  {
   recovery_info->status |= REC_INFO_FAILOVER_FAIL;
   goto exit_point;
  }
 }
 else
 {
  rec__boot_recover_dbs(backout_ctx, rec_caller_arg,
        SERIAL);
 }
}
exit_point:
/*
** Release lock on model as recovery is complete
*/
if (rec_caller_arg->model_recovered)
{
 backout_ctx->model_locked = FALSE;
 dbt_unkeep(rec_caller_arg->modeldbt);
 dbt_unlock(rec_caller_arg->modeldbt);
}
/*
** Clear the P_ISRECOVERY bit in the pss as we are done recovering
** the databases
*/
pss->pstat &= ~(P_ISRECOVERY);
/*
** restore the old buffer pool configuration according to what
** we remembered in the rec_order_info structure, if the global
** status indicates that we need to.
```

```
 */
 if (recovery_info->status & REC_INFO_CACHE_NEEDRESTORE)
 {
  rec__tune_bufpools(REC_RESTORE_CONFIG, DEFAULT_CACHE_ID);
  recovery_info->status &= ~(REC_INFO_CACHE_NEEDRESTORE);
 }
 return;
}
/*
** REC__TUNE_BUFPOOLS
**
** Type: internal function (called by rec_run_parallel_recovery)
**
** Purpose:
** According to the passed in action, do different work.
** 1. REC_SET_CONFIG:  reconfigure the buffer pools in the
**    cache whose cache id is passed in to
**    optimize recovery performance.
** 2. REC_RESTORE_CONFIG: restore the old configuration for the buffer
**    pools to the original values before recovery
**    changed them.
**
** Parameters:
** action - the action to take. The allowed choices are:
**  REC_SET_CONFIG,  REC_RESTORE_CONFIG.
** cid - the cache id of the cache to tune.
**
** Returns:
** None.
**
** Side Effects:
** -- For REC_SET_CONFIG, the default pool and the pool with largest
**    possible mass size in the cache are reconfigured:
**    1. size of largest mass pool = REC_TUNE_POOL_RATE *
**  (size of largest mass pool + size of default pool)
**    For example, on a 2k server, before tuning, the cache has the
**    following pools:
**  16K pool = 200M default pool = 500M, and there may
**  be some other pools in the cache which will not be touched
**  by us.
**    After tuning, the pools become (assume REC_TUNE_POOL_RATE = 0.4):
**  16K pool = (500 + 200) * 0.4 = 280M
**  2K pool = (500 + 200) - 280 = 420M
**    2. the "async prefetch limit" (i.e. apf) in both pools are
**    configured to REC_OPTIMAL_APF.
**
** -- For REC_RESORE_CONFIG, the old configuration values for the
** tuned buffer pools will be restored.
**
** NOTE:
```

```
** 1. If the buffer pools have DEFAULT as the apf value, the value
** of "global async prefetch limit" will be used. Then a possible race
** between recovery tuning process reading the value and another client
** trying to change the global value via sp_configure.
**
** Considering the possibility of such a situation and the complexity to
** the code to handle the synchronization completely, it will be handled
** only at a basic level:
** A check will be added to sp_configure to make sure that this config
** parameter cannot be changed if the server is in recovery state.
**
**  2. changing of "global cache partition number" during recovery could
** cause recovery to fail to restore the original sizes of the pools
** because the number of masses to move may no longer be divisible by
** the new number of cache partitions in the cache, and sp_poolconfig
** will then not move the exact amount of memory that recovery requires.
**
** Thus, we will not allow change to this config parameter during recovery
** either.
**
** Both of the above restrictions will be documented.
**
** Synchronization:
** The synchronization is handled in the following ways:
** 1. in the stored procedure sp_poolconfig: The server recovery state
** (@@srvr_rec_state) will be checked. If the server is in recovery
** (failover or boot time), no other client can change the pool
** configuration except for recovery itself.
**
** 2. REC_INFO_BOOTTIME_RECOVERY and REC_INFO_FAILOVER_RECOVERY will be
** used as the secondary barrier. In cfg__cache_pool_config(), after the
** transit lock is aquired on the cache to change the pool configurations,
** server recovery state will be checked. If either bits is set, and it
** is not the recovery process, the configuration will fail.
**
** 3. Since the cache is locked by the cache transit lock while the
** pool configuration is changed and is released after the changes are
** done, before doing any work, rec__tune_bufpools will try to acquire
** the lock. After it gets the lock, it can release it because now
** the recovery bit will be blocking other clients from changing the
** configuration.
** The last barrier is to handle situations where client has already
** passed the check point for the status bits, and is still doing the
** change when recovery process come to this function.
**
** History:
** ('10 2002) - created (fzhou)
*/
SYB_STATIC void
rec__tune_bufpools(int action, cacheid_t cid)
```

```
{
 CACHE_PTR cacheptr;  /*
     ** Pointer to array of cachelet
     ** pointers.
     */
 CACHE_DESC *control_cachelet; /*
     ** the control cachelet for
     ** the cache
     */
 BUF_POOL_DESC *small_io_pool;  /* the buffer pool with the
     ** smallest mass size in the
     ** cache.
     */
 BUF_POOL_DESC *large_io_pool;  /* the buffer pool with the
     ** largest mass size
     */
 RECOVERY_INFO *recovery_info;  /* local pointer to the
     ** global structure.
     */
 CFG_HEAD *cfginfo;  /* pointer to
     ** Resource->rcfg_info.
     */
 CACHE_DESC *cfg_rec_cache;  /* pointer to the cache
     ** descriptor allocated in
     ** Resource->rcfg_info to hold
     ** the original config for the
     ** cache before tuning.
     */
 unsigned int cache_size;  /* size of the recovery cache,
     ** in K.
     */
 size_t  small_mass_pool_size; /* total memory size in the
     ** smallest io pool (in kbytes).
     */
 size_t  large_mass_pool_size; /* total memory size in the
     ** largest io pool (in kbytes).
     */
 int32  largest_io_size; /* The largest io size (in
     ** kbytes).
     */
 int32  largest_io_size_in_bytes;
     /* largest io size in bytes. */
 size_t  new_large_pool_size; /* amount of memory to configure
     ** for the largest io pool
     ** (in Kbytes).
     */
 size_t  new_tot_masses;  /* total number of masses
     ** in the largest io pool
     ** after the reconfig.
     */
```

```
int  num_cachelets;
size_t  bytes_to_move;
size_t  num_masses_to_move; /* number of masses in the
    ** destination pool that will
    ** be moved.
    */
int32  new_apf_value;
int32  old_apf_value;  /* the old apf value for
    ** print.
    */
int32  global_apf_value; /* store the global apf percent
    ** value for comparison if
    ** the local apf percent for
    ** the pool is set to DEFAULT.
    */
char  cmdbuf[SHORTTEXT];
char  new_apf_value_str[MAXSIZESTR];
char  smallest_io_size_str[MAXSIZESTR];
    /* store the string
    ** representation of the mass
    ** size of the small mass pool
    */
char  largest_io_size_str[MAXSIZESTR];
    /* store the string
    ** representation of the mass
    ** size of the large mass pool
    */
char  new_large_pool_size_str[MAXSIZESTR];
    /* store the string
    ** representation of the
    ** desired pool size for
    ** the destination buffer pool.
    ** This pool could be the
    ** small mass pool or the
    ** large mass pool, depending
    ** on the pool sizes.
    */
pgsz_t  server_pagesize;
SYB_BOOLEAN poolsize_changed;
SYB_BOOLEAN use_global_apf;
int  lockstat;
LOCALPSS(pss);
/* keep these backout variables in memory */
VOLATILE struct
{
 PSS *pss;
} copy;
SYB_NOOPT(copy);
/* Initializations */
copy.pss = pss;
```

```
/*
** Install a handler here to catch all exceptions, print out the
** error message and just return to the caller.
*/
if (ex_handle(EX_ANY, EX_ANY, EX_ANY, hdl_backout_msg))
{
 pss  = copy.pss;
 pss->p5stat &= ~(P5_REC_TUNE_CACHE);
 return;
}
/* Initialize local variables. */
poolsize_changed = use_global_apf = FALSE;
global_apf_value = UNUSED;
/* Get the current page size for the server */
server_pagesize = CFG_GETCURVAL(cmaxdbpagesize);
recovery_info = &Resource->rrecovery_info;
/*
** Set the bit to indicate we are tuning the cache.  Config manager
** relies on this to block users from reconfiguring the default data
** cache during recovery and allows only the recovery thread to do so
*/
pss->p5stat |= P5_REC_TUNE_CACHE;
/*
** default data cache is the cache that recovery will use, and thus
** needs tuning. Get the cache from Resource->rcaches which stores
** the run values for the caches.
**
** Since the functions we call to reconfigure the cache will loop
** through all cachelets in this cache and tune the buffer pools,
** we should consider the cache as a whole in terms of how
** to tune it.  Consult the control cachlet for the information
** regarding pool size and apf percentage.
**
** Since we have made sure that no one else can change the
** configuration of this cache during recovery, we don't have
** use the spinlock to access the cache.
*/
CM_GET_CACHEPTR_CID(cid, cacheptr);
control_cachelet = cacheptr[CNTRL_CACHELET];
/*
** Before we get any current values, try to acquire the
** cache transit lock.
**
** The lock we acquire will be released instantly,
** because the status bit REC_INFO_BOOTTIME_RECOVERY or
** REC_INFO_FAILOVER_RECOVERY will be used to prevent any changes
** to the cache configuration from other clients.
*/
lockstat = LOCK_CACHE(LOCK_INSTANT, EX_ADDR, cacheptr);
/* If we didn't get a lock, do not change the configuration. */
```

```
if (lockstat < 0)
{
 mnt_ex_print(EX_NUMBER(RECOVER2, REC_CANT_GET_LOCK), EX_INTOK, 1,
    pss->pspid,
    control_cachelet->clen,
    control_cachelet->cname);
 /* Clear the status bit indicating that we are tuning */
 pss->p5stat &= ~(P5_REC_TUNE_CACHE);
 return;
}
/*
** Get the size of the cache which is to be tuned. We need to
** subtract coverhead from ctotalcachesize to get the available
** size (in K).
*/
cache_size = control_cachelet->ctotalcachesize -
    control_cachelet->coverhead;
/* Get the number of cache partitions for this cache. */
num_cachelets = NUM_CACHELETS(cacheptr);
/*
** RESOLVE:
** Currently (as of 12.5), the run size information of pool size
** and wash size in the control cachelet donot necessarily reflect
** the correct run size of the configurations if the cache has
** more than 1 cache partitions.
** I.e.
** control_cachelet->pool->btotal_masses !=
**   (cachelet(1)->pool->btotal_masses +...
**    cachelet(n)->pool->btotal_masses)
** when control_cachelet->pool->btotal_masses is not divisible by
** the number of cachelets which is denoted by "n" in the above
** statement.
**
**  The reason for this is:
** 1. the size of each cachelet is always the same, and it is
** calculated by (btotal_masses / num_cachelets);
** 2. control_cachelet->pool->btotal_masses is calculated based
** on the total memory in the pool. Even if the number is not
** divisible by the number of cachelets, it is not adjusted.
**
** In such case, if recovery tries to restore the original
** configuration value for the pools, which is obtained from the
** control cachelet, server will only be able to restore the total
** size of the pool to the closest number divisible by the number
** of cachelets.
**
**  It is currently being considered to fix the information in
** the control cachelet to reflect the real run size.
*/
/*
```

```
** Now we need to get the pools that we are interested in,
** i.e. the pool with the smallest mass size (the pool mass size
** equals to the page size), and that with the largest mass size.
**
** Since memory for pool descriptors is always preallocated
** whether or not there is any buffers in a certain size pool,
** the pool descriptors will always be not NULL.
*/
small_io_pool = control_cachelet->cpools[BUF_POOL_0];
large_io_pool = control_cachelet->cpools[BUF_POOL_3];
/*
** Convert the pool mass sizes for both buffer pools into
** string. These will be used in the internal sql call
** to the stored procedure to move buffers between the pools.
*/
snprintf(smallest_io_size_str, sizeof(smallest_io_size_str), "%d",
  (int) KUNITS_FROM_BYTES(small_io_pool->bpoolmsize));
strcat(smallest_io_size_str, "K");
/*
** Cannot use large_io_pool->bpoolmsize here because if the
** pool doesn't exist, this field will be 0. Instead, use the
** largest mass size supported in the server.
*/
largest_io_size_in_bytes = MAX_MASS_SIZE_CUR_SUPP(server_pagesize);
largest_io_size = KUNITS_FROM_BYTES(largest_io_size_in_bytes);
snprintf(largest_io_size_str, sizeof(largest_io_size_str), "%d",
  (int) largest_io_size);
strcat(largest_io_size_str, "K");
/*
** Get the cache config block which is used to store the original
** cache configuration for default data cache.
*/
cfginfo = cfg_getmain();
cfg_rec_cache = cfginfo->cfg_rec_cache[DEFAULT_CACHE_ID];
/*
** If the action is REC_SET_CONFIG, we will want to have the
** large_mass_pool_size /
**  (small_mass_pool_size + large_mass_pool_size)
** equals to a fixed value. This fixed value is set by ASE
** internally as REC_TUNE_POOL_RATE.
*/
if (action == REC_SET_CONFIG)
{
 /*
 ** RESOLVE: If the sum of the small_mass_pool_size
 ** and the large_mass_pool_size is less than 50% of the
 ** total cache size, do we borrow buffers from other
 ** buffer pools to make the sum of the two pools that will
 ** be used by recovery big enough (for example, 70% of the
 ** total cache size)?
```

```
**
** Apart from applications may use the pool whose mass size
** is twice the default mass size for log, is there any other
** reason why we shouldn't borrow buffers from other pools?
*/
/*
** Initialize the fields which will be used to store the
** before-change values.
*/
large_io_pool->bold_apf_percent =
 small_io_pool->bold_apf_percent = VALUE_NOT_CHANGED;
/*
** Clear the status that indicates the pool size has changed.
*/
large_io_pool->bstatus &= ~(BPOOL_SIZE_CHANGED);
small_io_pool->bstatus &= ~(BPOOL_SIZE_CHANGED);
/*
** Remember the original cache configuration in the
** Resource->rcfg_info->cfg_rec_cache, which was
** allocated during start up time.
*/
cfg__zerocache(cfginfo, cfg_rec_cache);
/*
** Copy the configuration information for the default
** data cache before recovery changes it.
** When a config file is written out during recovery,
** the saved information will be used to write out
** pool configuration in the default data cache because
** what recovery has changed internally shouldn't be
** visible in the config file.
*/
cfg__copycache(cfg_rec_cache,
   cfginfo->cfgcache[DEFAULT_CACHE_ID]);
/*
** Get the pool size for both buffer pools by multiplying the
** number of masses in the pool by the size of the mass.
** At start up time, usually the value in btotal_init and
** btotal_masses would be the same, but just in case the buffer
** pool sizes were changed before we come here, We use
** btotal_masses, which will always be the up-to-date size of
** the buffer pool.
**
** The sizes are round up to KUNITS from BYTES.
*/
small_mass_pool_size = KUNITS_FROM_BYTES(
   ((size_t)small_io_pool->btotal_masses) *
   small_io_pool->bpoolmsize );
large_mass_pool_size = KUNITS_FROM_BYTES(
   ((size_t)large_io_pool->btotal_masses) *
   large_io_pool->bpoolmsize );
```

```
/*
** The distribution of the borrowed buffers will change
** the rate of new_small_pool_size vs. new_large_pool_size
** to a fixed value which can help the recovery
** performance.
*/
new_large_pool_size =
 (large_mass_pool_size + small_mass_pool_size)
    * REC_TUNE_POOL_RATE;
/*
** Get the total number of masses in the largest io pool
** with the new memory size.
** The denominator is the number of K per mass.
*/
new_tot_masses = new_large_pool_size /
      (BUFS_IN_MASS(largest_io_size_in_bytes,server_pagesize) *
      KUNITS_FROM_BYTES(server_pagesize));
/*
** Some validations to the new memory size for the
** largest io pool:
** If any of the failure conditions holds true, directly
** go to the next tuning step, which is to tune the apf value.
**
** 1. the new value is the same as the existing
** configuration value.
*/
if (new_large_pool_size == large_mass_pool_size)
{
 if (TRACECMDLINE(RECOVER, 76))
 {
  scerrlog("No need to reconfigure the pool size for the '%.*s' pool, because it is already what we want: %dK.\n",
    sizeof(largest_io_size_str),
    largest_io_size_str,
    large_mass_pool_size);
 }
 /*
 ** Set the status bit indicating that recovery
 ** can use largest io pool.
 */
 recovery_info->status |=
     REC_INFO_USE_LARGEST_IO_POOL;
 goto tune_apf;
}
/*
** 2. the new amount of memory for the largest io pool
** is less than the minimum amount of memeory required
** for a buffer pool.
*/
if (((new_tot_masses / num_cachelets) < MIN_MASSES_IN_POOL) ||
    ((new_large_pool_size / num_cachelets) <
  (int)(BUFPOOL_LOW_WATER(server_pagesize))) )
{
 if (TRACECMDLINE(RECOVER, 76))
 {
  scerrlog("The new memory for largest io pool %dK (%d buffers) is too small.\n",
   new_large_pool_size, new_tot_masses);
 }
 goto tune_apf;
}
/*
** 3. If the amount of memory left for the smallest io
** pool (i.e. default pool) is less than the minimum amount
** of memeory required for a buffer pool.
**
** Since the mass size for the default pool is the same as
** the buffer size, checking the BUFPOOL_LOW_WATER
** should be enough (see the definition of this macro for
** detail).
*/
if ((large_mass_pool_size +
 small_mass_pool_size - new_large_pool_size) <
  (int)(BUFPOOL_LOW_WATER(server_pagesize)) )
{
 if (TRACECMDLINE(RECOVER, 76))
 {
  scerrlog("The new memory %dK for smallest io pool is too small.\n",
   (large_mass_pool_size +
   small_mass_pool_size
   - new_large_pool_size));
 }
 goto tune_apf;
}
/*
** 4. Check to see if the memory to move is valid, i.e. it
** is at least big enough for (one mass * number of cachelets)
** in the largest io pool, Because masses are distributed
** equally to each cachelet.
*/
/* The memory to move in bytes. */
bytes_to_move = BYTES_FROM_KUNITS(abs(new_large_pool_size -
     large_mass_pool_size));
num_masses_to_move = bytes_to_move /
 (BUFS_IN_MASS(largest_io_size_in_bytes,server_pagesize)
  * server_pagesize);
/*
** Divide the num_masses_to_move by the number of cachelets.
*/
num_masses_to_move = num_masses_to_move / num_cachelets;
/* The number of masses to move must be no less than 1. */
if (num_masses_to_move < 1)
```

```
 {
  if (TRACECMDLINE(RECOVER, 76))
  {
   scerrlog("The amount of memory to move %d is too small. It must be at least big enough for %d %dK mass.\n",
     bytes_to_move,
     num_cachelets,
     largest_io_size);
  }
  /*
  ** Set the status bit indicating that recovery
  ** can use largest io pool.
  */
  recovery_info->status |=
      REC_INFO_USE_LARGEST_IO_POOL;
  goto tune_apf;
 }
 /*
 ** To avoid failure to move all memory due to the total
 ** number of masses to move not divisible by the number
 ** of cachelets, round down the new large pool size.
 ** If trace flag 3473 is on, don't do this.
 */
 if (!TRACECMDLINE(RECOVER, 73) && (num_cachelets != 1))
 {
  if (new_large_pool_size > large_mass_pool_size)
  {
   new_large_pool_size = large_mass_pool_size +
     (num_masses_to_move *
      num_cachelets *
      largest_io_size);
  }
  else
  {
   new_large_pool_size = large_mass_pool_size -
    (num_masses_to_move *
     num_cachelets *
     largest_io_size);
  }
 }
 if (TRACECMDLINE(RECOVER, 76))
 {
  scerrlog("We will try to move %d masses per cachelet to make the %dK pool have %dK memory. Its original size is
%dK.\n",
     num_masses_to_move,
     largest_io_size,
     new_large_pool_size,
     large_mass_pool_size);
 }
 /*
 ** Before changing the values, set the bit in the pools to
** remember this fact.
*/
large_io_pool->bstatus |= BPOOL_SIZE_CHANGED;
small_io_pool->bstatus |= BPOOL_SIZE_CHANGED;
/*
** Set the bit indicating that we will need to restore
** the original configurations in the buffer pools.
*/
recovery_info->status |= REC_INFO_CACHE_NEEDRESTORE;
/*
** Use internal sql call to adjust the buffer
** pools to the optimal rate between the
** largest mass pool and the default mass pool.
** I.e. set the size of the pool with
** largest mass size to the optimal size that
** we have determined (the config_pool will be
** the large_io_pool, and the affected pool
** will be the small_io_pool). We don't have
** to worry about the direction of buffer
** movement, as the stored procedure and the
** configuration functions will take care of it.
*/
snprintf(new_large_pool_size_str,
 sizeof(new_large_pool_size_str), "%d",
 (int) new_large_pool_size);
strcat(new_large_pool_size_str, "K");
snprintf(cmdbuf, sizeof(cmdbuf),
  "sp_do_poolconfig \'%s\', \'%s\', \'%s\', \'%s\', \'true\'",
  control_cachelet->cname,
  new_large_pool_size_str,
  largest_io_size_str,
  smallest_io_size_str);
poolsize_changed = INTERNAL_SQL_BASIC((BYTE *) cmdbuf);
/*
** If we didn't reconfigure the size of the buffer
** pools due to some error in internal_sql, clear the
** bits indicating pool size change.
** Sometimes, errors in the stored procedure can't be
** detected by the internal_sql. So check for the size
** of the large_io_pool. Consider the action failed if
** the large_io_pool was not created.
*/
if ((poolsize_changed == FALSE) ||
  (large_io_pool->btotal_masses == 0))
{
 large_io_pool->bstatus &= ~(BPOOL_SIZE_CHANGED);
 small_io_pool->bstatus &= ~(BPOOL_SIZE_CHANGED);
 recovery_info->status &=
   ~(REC_INFO_CACHE_NEEDRESTORE);
 mnt_ex_print(EX_NUMBER(RECOVER2, REC_CANT_TUNE_POOLSIZE), EX_INFO, 1,
      control_cachelet->clen,
      control_cachelet->cname);
   }
   else
   {
    /*
    ** Set the status bit indicating that recovery
    ** can use largest io pool.
    */
    recovery_info->status |=
        REC_INFO_USE_LARGEST_IO_POOL;
    mnt_ex_print(EX_NUMBER(RECOVER2, REC_TUNE_BUFPOOL), EX_INFO, 1,
      sizeof(largest_io_size_str),
      largest_io_size_str,
      control_cachelet->clen,
      control_cachelet->cname);
    mnt_ex_print(EX_NUMBER(RECOVER2, REC_TUNE_BUFPOOL), EX_INFO, 2,
      sizeof(smallest_io_size_str),
      smallest_io_size_str,
      control_cachelet->clen,
      control_cachelet->cname);
   }
   /*
   ** TESTING POINT: If trace flag 3472 is on, raise a fatal
   ** error half way through the tuning of buffer pools.
   */
   if (TRACECMDLINE(RECOVER, 72))
   {
    ex_raise(RECOVER, REC_RETURN, EX_CMDFATAL, 7);
   }
  tune_apf:
   /*
   ** Go ahead and tune the apf in the available pools for
   ** recovery. We will tune the pool with largest mass
   ** and the the default pool if they are not the same.
   */
   new_apf_value = REC_OPTIMAL_APF;
   snprintf(new_apf_value_str, sizeof(new_apf_value_str),
    "%d", (int) new_apf_value);
   /*
   ** We only tune the apf value in the default pool and
   ** the pools with largest possible io size on the server.
   **
   ** Tune the apf percent in the large_io_pool only
   ** when recovery will use such a pool. If this bit is
   ** not set, recovery will use the default pool for both
   ** log and data i/o with optimized apf value.
   */
   if (recovery_info->status & REC_INFO_USE_LARGEST_IO_POOL)
   {
```

```
/*
** Assert that the large io pool exists.
*/
SYB_ASSERT((large_io_pool->bpoolmsize != 0) &&
    (large_io_pool->btotal_masses != 0));
/*
** If the current value is DEFAULT, make sure that
** we use the global apf percent for checking.
*/
if (large_io_pool->bapf_percent ==
   APF_UNSPECIFIED_OR_DEFAULT)
{
 use_global_apf = TRUE;
 global_apf_value =
     Resource->rconfig->cfg_global_apf_percent;
}
/*
** Tune the apf value if it is not already
** what recovery needs. If the pool doesn't have
** a local apf specified, need to check if the
** global apf limit value is what recovery needs.
*/
if ( ((!use_global_apf) &&
      (large_io_pool->bapf_percent != new_apf_value)) ||
     ((use_global_apf) &&
      (global_apf_value != new_apf_value)) )
{
 /*
 ** Before calling bufupdateapfpercent()
 ** to update, store the old value in the pool.
 */
 large_io_pool->bold_apf_percent =
  large_io_pool->bapf_percent;
 /*
 ** Set the bit indicating that we will need to
 ** restore the original configurations in the
 ** buffer pools.
 ** This bit might have already been set, but
 ** set it again anyway.
 */
 recovery_info->status |=
   REC_INFO_CACHE_NEEDRESTORE;
 /*
 ** The return value of bufupdateapfpercent()
 ** should always be TRUE, as the only case
 ** where it would return FALSE is that the new
 ** apf_percent is out of range, which won't be
 ** our case.
 ** Therefore, no need to check the return value.
 */
```

```
  (void) bufupdateapfpercent( cid,
   largest_io_size_in_bytes,
   new_apf_value,
   new_apf_value_str);
  /*
  ** If the local apf value for the pool was
  ** set to DEFAULT, print the global apf value
  ** as the old value instead of the defined
  ** value for APF_UNSPECIFIED_OR_DEFAULT
  ** (i.e. -1).
  */
  old_apf_value = (use_global_apf)?
    global_apf_value :
    large_io_pool->bold_apf_percent;
  mnt_ex_print(EX_NUMBER(RECOVER2, REC_TUNE_APF), EX_INFO, 1,
   sizeof(largest_io_size_str),
   largest_io_size_str,
   control_cachelet->clen,
   control_cachelet->cname,
   old_apf_value, new_apf_value);
 }
}
/*
** Now we will tune the apf in the default pool.
*/
use_global_apf = FALSE;
/*
** Similarly, check to see if there is any local apf value
** specified for this pool. If not, need to use the global
** apf value later for checking.
*/
if (small_io_pool->bapf_percent == APF_UNSPECIFIED_OR_DEFAULT)
{
 use_global_apf = TRUE;
 global_apf_value =
  Resource->rconfig->cfg_global_apf_percent;
}
/*
** Tune the apf for the default pool only when its
** current value is not the same as what recovery needs.
*/
if ( ((!use_global_apf) &&
          (small_io_pool->bapf_percent != new_apf_value)) ||
     ((use_global_apf) && (global_apf_value != new_apf_value)) )
{
 /*
 ** Before calling bufupdateapfpercent()
 ** to update, store the old value in the pool.
 */
 small_io_pool->bold_apf_percent =
  small_io_pool->bapf_percent;
 /*
 ** Set the bit indicating that we will need to
 ** restore the original configurations in the
 ** buffer pools.
 ** This bit might have already been set, but
 ** set it again anyway.
 */
 recovery_info->status |=
   REC_INFO_CACHE_NEEDRESTORE;
 /*
 ** For the same reason as stated before the previous
 ** call to this function, we don't need to check the
 ** return value.
 */
 (void) bufupdateapfpercent(DEFAULT_CACHE_ID,
   server_pagesize, new_apf_value,
   new_apf_value_str);
 /*
 ** If the local apf value for the pool was
 ** set to DEFAULT, print the global apf value
 ** as the old value instead of the defined
 ** value for APF_UNSPECIFIED_OR_DEFAULT
 ** (i.e. -1).
 */
 old_apf_value = (use_global_apf) ?
   global_apf_value :
   small_io_pool->bold_apf_percent;
 mnt_ex_print(EX_NUMBER(RECOVER2, REC_TUNE_APF), EX_INFO, 2,
   sizeof(smallest_io_size_str),
   smallest_io_size_str,
   control_cachelet->clen,
   control_cachelet->cname,
   old_apf_value, new_apf_value);
}
/*
** If none of the configurations in the two pools has
** been changed, make sure that the REC_INFO_CACHE_NEEDRESTORE
** bit is not set.
*/
if (REC_POOL_NOT_TUNED(large_io_pool) &&
   REC_POOL_NOT_TUNED(small_io_pool))
{
 if (TRACECMDLINE(RECOVER, 76))
 {
  scerrlog("Recovery didn't tune '%.*s'.\n",
    control_cachelet->clen,
    control_cachelet->cname);
 }
 /*
```

```
  ** Since from the code path, we assured that
  ** this bit won't be set if nothing has changed,
  ** put an assertion here for sanity check.
  */
  SYB_ASSERT(!(recovery_info->status &
     REC_INFO_CACHE_NEEDRESTORE));
  /*
  ** Although this is not necessary, clear the bit
  ** here just to be safe.
  */
  recovery_info->status &= ~(REC_INFO_CACHE_NEEDRESTORE);
 }
}
/* If we are to restore the old buffer pool configuration... */
else
{
 SYB_ASSERT(action == REC_RESTORE_CONFIG);
 /*
 ** Since recovery only tunes the default pool and
 ** the pool with largest io allowed on the server,
 ** restore the configurations for the two pools.
 ** 1. The apf value of the largest io pool.
 */
 new_apf_value = large_io_pool->bold_apf_percent;
 if (new_apf_value != VALUE_NOT_CHANGED)
 {
  /*
  ** Since the apf value of the largest io
  ** pool was tuned by recovery, assert that
  ** recovery had used the large io pool.
  */
  SYB_ASSERT(recovery_info->status &
    REC_INFO_USE_LARGEST_IO_POOL);
  old_apf_value = large_io_pool->bapf_percent;
  /*
  ** If the pool had DEFAULT as the local apf
  ** value, pass APF_UNSPECIFIED_OR_DEFAULT as
  ** the apf_percent to bufupdateapfpercent, which
  ** will set the apf value in the pool back to
  ** DEFAULT.
  */
  if (new_apf_value ==
    APF_UNSPECIFIED_OR_DEFAULT)
  {
   sprintf(new_apf_value_str, "%s", "DEFAULT");
  }
  else
  {
   snprintf(new_apf_value_str,
    sizeof(new_apf_value_str),
   "%d", (int) new_apf_value);
  }
  (void) bufupdateapfpercent(cid,
   largest_io_size_in_bytes,
   new_apf_value,
   new_apf_value_str);
  mnt_ex_print(EX_NUMBER(RECOVER2, REC_RESTORED_POOLAPF), EX_INFO, 2,
    sizeof(largest_io_size_str),
    largest_io_size_str,
    control_cachelet->clen,
    control_cachelet->cname,
    old_apf_value,
    sizeof(new_apf_value_str),
    new_apf_value_str);
  /*
  ** clear the field with old apf value because
  ** we have successfully restored it.
  */
  large_io_pool->bold_apf_percent = VALUE_NOT_CHANGED;
 }
 /*
 ** 2. Restore apf percent for DEFAULT pool.
 */
 new_apf_value = small_io_pool->bold_apf_percent;
 if (new_apf_value != VALUE_NOT_CHANGED)
 {
  old_apf_value = small_io_pool->bapf_percent;
  /*
  ** If the pool had DEFAULT as the local apf
  ** value, pass APF_UNSPECIFIED_OR_DEFAULT as
  ** the apf_percent to bufupdateapfpercent, which
  ** will set the apf value in the pool back to
  ** DEFAULT.
  */
  if (new_apf_value ==
    APF_UNSPECIFIED_OR_DEFAULT)
  {
   sprintf(new_apf_value_str, "%s", "DEFAULT");
  }
  else
  {
   snprintf(new_apf_value_str,
    sizeof(new_apf_value_str),
    "%d", (int) new_apf_value);
  }
  (void) bufupdateapfpercent(cid,
    server_pagesize,
    new_apf_value,
    new_apf_value_str);
  mnt_ex_print(EX_NUMBER(RECOVER2, REC_RESTORED_POOLAPF), EX_INFO, 2,
 sizeof(smallest_io_size_str),
 smallest_io_size_str,
 control_cachelet->clen,
 control_cachelet->cname,
 old_apf_value, sizeof(new_apf_value_str),
 new_apf_value_str);
/*
** clear the field with old apf value because
** we have successfully restored it.
*/
small_io_pool->bold_apf_percent = VALUE_NOT_CHANGED;
}
/*
** If we have changed the buffer pool size, restore the    ** old size from the saved information in the cfg_rec_cache.
*/
if (large_io_pool->bstatus & BPOOL_SIZE_CHANGED)
{
 /*
 ** If the pool didn't exist, delete it by setting
 ** the size to 0.
 */
 if (cfg_rec_cache->cpools[BUF_POOL_3]->bvaluestr[0]
       == '\0')
 {
  sprintf(new_large_pool_size_str, "0K");
 }
 else
 {
  strncpy(new_large_pool_size_str,
   cfg_rec_cache->cpools[BUF_POOL_3]->bvaluestr,
   sizeof(new_large_pool_size_str));
 }
 if (TRACECMDLINE(RECOVER, 76))
 {
  scerrlog("We will try to restore the %d pool to have %s memory.\n",
   large_io_pool->bpoolmsize,
   new_large_pool_size_str);
 }
 snprintf(cmdbuf, sizeof(cmdbuf),
   "sp_do_poolconfig \'%s\', \'%s\', \'%s\', \'%s\', \'true\'",
   control_cachelet->cname,
   new_large_pool_size_str,
   largest_io_size_str,
   smallest_io_size_str);
 poolsize_changed =  INTERNAL_SQL_BASIC((BYTE *) cmdbuf);
 if (poolsize_changed == TRUE)
 {
  mnt_ex_print(EX_NUMBER(RECOVER2, REC_RESTORED_POOLSIZE), EX_INFO, 1,
   sizeof(largest_io_size_str),
   largest_io_size_str,
    sizeof(smallest_io_size_str),
    smallest_io_size_str,
    control_cachelet->clen,
    control_cachelet->cname);
  }
  /*
  ** What if we couldn't restore the old configuration?
  ** Should we retry?
  ** For now, print a message.
  */
  else
  {
   mnt_ex_print(EX_NUMBER(RECOVER2, REC_CANT_RESTORE_BUFPOOL), EX_INFO, 1,
    sizeof(largest_io_size_str),
    largest_io_size_str,
    control_cachelet->clen,
    control_cachelet->cname,
    sizeof(new_large_pool_size_str),
    new_large_pool_size_str);
  }
 }
 /* Clear the bits indiating pool configurations have changed. */
 large_io_pool->bstatus &= ~(BPOOL_SIZE_CHANGED);
 small_io_pool->bstatus &= ~(BPOOL_SIZE_CHANGED);
 recovery_info->status &=
   ~(REC_INFO_CACHE_NEEDRESTORE);
}
/* Clear the status bit indicating that we are tuning */
pss->p5stat &= ~(P5_REC_TUNE_CACHE);
return;
}
/*
** REC__PARALLEL_HDLR
**
** This is the entry function invoked by each spawned recovery threads.
**
** Purpose:
** 1. At the beginning of the function, Set up the pss for recovery;
**
** 2. Call the main entry function rec__boot_recover_dbs to do the
**  actual job. Call it with PARALLEL as the recovery mode.
**
** 3. if all dbs are recovered, or if the server was set to go back to
** serial recovery, wake up initial recovery thread.
**
** 4. Before exit, cleanup pss and kill itself.
**
** Parameters:
** passed_in_arg - The rec_caller_arg structure that was set up by the
**   caller which contains the informaion we need for
**   lower recovery function.
**
** Returns:
** None.
**
** History:
** (5'02 fzhou) - created
*/
SYB_STATIC void CDECL
rec__parallel_hdlr(FNIPARAM passed_in_arg)
{
 engid_t  engine_num;
 SYB_BOOLEAN failover_recovery;
 SYB_BOOLEAN boot_recovery;
 SYB_BOOLEAN parallel_recovery;
 REC_CALLER_ARG *rec_caller_arg;
 LOCALPSS(pss);
 VOLATILE REC_CALLER_BKOUT_CTX backout_ctx;
 /* keep backout variables im memory */
 SYB_NOOPT(backout_ctx);
 MEMZERO(&backout_ctx, sizeof(backout_ctx));
 ex_init(pss);
 INSTALL_BACKOUT_HANDLES(&pss->pbkout_rec_caller, &backout_ctx,
      rec_caller_backout);
 /*
 ** Set the global_cleanup indicator to TRUE because if any
 ** error happens, we want to clean up the resource we acquires
 ** at the end of rec__caller_hdlr().
 */
 backout_ctx.global_cleanup = TRUE;
 if (ex_handle(EX_ANY, EX_ANY, EX_ANY, rec__caller_hdlr))
 {
  /*
  ** will not return because the cleanup function which is
  ** called by the rec__caller_hdlr() will call quitfn()
  ** to kill this thread.
  */
  return;
 }
 /* Initialize local variables. */
 parallel_recovery =FALSE;
 boot_recovery = TRUE;
 failover_recovery = IS_FAILOVER_THREAD();
 rec_caller_arg = (REC_CALLER_ARG *)passed_in_arg;
 if (failover_recovery)
 {
  /*
  ** For now, only failover recovery and boot time recovery
  ** will use parallel recovery. So set boot_recovery to
  ** FALSE if failover_recovery is TRUE.
 */
 boot_recovery = FALSE;
}
/*
** Set the process type in pss for spawned recovery threads
** so that in the exception clean up code, the spawned
** threads could be distinguished fron the initial thread.
*/
pss->pprocess_type = RECOVERY;
/* mark Pss so we can tell who is recovery */
pss->pstat |= P_ISRECOVERY;
/*
** Mark PSS as 'no log suspend' so that crossing the log's
** last-chance threshold will raise an 1105 rather than
** suspending this task.
*/
pss->p2stat |= P2_NOLOGSUSPEND;
DEBUG_SET(pss);
/* Set the status if it is a boot time recovery process */
if (boot_recovery)
 pss->p3stat |= P3_BOOT_RECOVERY;
/*
** TESTING POINT: If traceflag 3469 is turned on, terminate the thread.
** This traceflag is used to simulate the failure to spawn a thread.
** Hence, we terminate the spawned thread gracefully here
*/
if (TRACECMDLINE(RECOVER, 69))
{
 scerrlog("TESTING: Simulating failure condition where we fail to spawn a recovery thread.  Hence, we are terminating
the spawned thread \n");
 terminate_process(pss->pkspid, 0);
}
/*
** WHen trace flag 3455 is on, affiniate the thread with the passed
** in engine.
*/
if (TRACECMDLINE(RECOVER, 55))
{
 engine_num = rec_caller_arg->engine_num;
 TRACEPRINT("Push affinity for spid %d to engine %d.\n",
  pss->pspid, engine_num);
 uppushaffinity(engine_num);
}
if (boot_recovery)
{
 rec__boot_recover_dbs((REC_CALLER_BKOUT_CTX *)&backout_ctx,
     rec_caller_arg, PARALLEL);
}
else if(failover_recovery)
{
  if (rec_failover_recover_dbs(
   (REC_CALLER_BKOUT_CTX *)&backout_ctx, PARALLEL) == FAIL)
  {
   /* set the status indicating that failover has failed */
   P_SPINLOCK(Resource->ha_spin);
   if (!Resource->rrecovery_info.status &
      REC_INFO_FAILOVER_FAIL)
   {
    Resource->rrecovery_info.status |=
      REC_INFO_FAILOVER_FAIL;
   }
   V_SPINLOCK(Resource->ha_spin);
  }
 }
 /*
 ** do necessary cleanup in pss.
 */
 pss->pstat &= ~P_ISRECOVERY;
 pss->p2stat &= ~P2_NOLOGSUSPEND;
 if (boot_recovery)
  pss->p3stat &= ~P3_BOOT_RECOVERY;
 (void)rec_thread_cleanup((FNIPARAM)pss, FALSE);
 /*
 ** will not return because quitfn() will be called by
 ** the cleanup function to kill this thread.
 */
 return;
}
/*
** BOOT_RECOVER_AND_ONLINE
**
** Type: internal function
**
** Purpose: This procedure performs the actual work to recover and
**  online a single database.
**
** Parameters:
** backout_ctx - the backout structure initialized in the caller
**   some fields will be filled up in this scope for
**   clean up.
**
** Return:
** None.
**
** History:
**  4/30/02 (fzhou) - written.
*/
SYB_STATIC void
boot__recover_and_online(REC_CALLER_BKOUT_CTX *backout_ctx)
{
```

```
SDES  *db_sdes;
BUF  *dbbuf;
short  datstat;
       BYTE     *stat3p;
int32  status3;
       int      stat3len;
int  ha_state;
dbid_t  curdbid;
SARG  keys[1];
DBTABLE  *dbt;
LOCALPSS(pss);
/*
** The database ID was stored in the context structure by the caller
** before we come here.
*/
curdbid = backout_ctx->curdbid;
/*
** The caller should only call us when there is a database needs
** to be recovered.
*/
SYB_ASSERT (curdbid != UNUSED);
/* open database  */
if (!usedb((BYTE *) NULL, curdbid, pss->psuid))
{
 /*
 ** In order to prevent databases from being marked suspect
 ** when we run out of DBTABLEs, since this is a resource
 ** error, raise an error with EX_RESOURCE severity, so that
 ** the database does not get marked suspect.  The database
 ** is left in an unrecovered state.
 ** Please see rec_dbsuspect() also
 */
 if ((EX_MAJOR(pss->plasterror) == OPENDBM) &&
  (EX_MINOR(pss->plasterror) == NO_DBTBLS))
 {
  ex_raise(RECOVER, REC_HALT_DBRECOVERY, EX_RESOURCE, 1,
     curdbid);
 }
 else
 {
  ex_raise(RECOVER, REC_RETURN, EX_CONTROL, 27);
 }
}
backout_ctx->backout_dbt = dbt = pss->pdbtable;
/* announce db being recovered */
ex_callprint(EX_NUMBER(RECOVER, REC_NAME), EX_INFO, 1,
 dbt->dbt_dbnlen, dbt->dbt_dbname);
/*
** Check if the devices are active and if so Recover the
** database
```

```
*/
if(!all_devices_active(DBT_DISKMAP(dbt)))
{
 /* print out message before freeing dbtable */
 ex_callprint(EX_NUMBER(RECOVER, REC_GIVEUP), EX_RESOURCE, 3,
  dbt, dbt->dbt_dbname, curdbid);
}
else if (!recovery(REC_INIT, (int *) UNUSED))
{
 /* print out message before freeing dbtable */
 ex_callprint(EX_NUMBER(RECOVER, REC_GIVEUP), EX_DBFATAL, 1,
  dbt->dbt_dbnlen, dbt->dbt_dbname, curdbid);
 /*
 ** Set the P_ISRECOVERY bit as the backout code could
 ** have cleared the bit in recovery()
 */
 pss->pstat |= P_ISRECOVERY;
}
else
{
 /*
 ** Recovery completed without error.
 **
 ** Clear DBT_NOTREC in the Sysdatabases row.
 ** Then clear it in the dbtable and
 ** close the database.
 */
 /* re-open sysdatabases temporarily */
 db_sdes = OPEN_SYSTEM_TABLE((objid_t) SYSDATABASES,
   MASTERDBID, Resource->rmasterdbt);
 /* register the sdes in backout structure */
 backout_ctx->db_sdes = db_sdes;
 db_sdes->sstat |= (SS_FGLOCK | SS_L1LOCK);
 /*
 ** TESTING POINT:
 ** If trace flag 3467 is on, raise a non-fatal
 ** error here.
 */
 if (TRACECMDLINE(RECOVER, 67))
 {
  ex_raise(RECOVER, REC_RETURN, EX_RESOURCE, 4);
 }
 initarg(db_sdes, keys, 1);
 setarg(db_sdes, &Sysdatabases[DAT_DBID], EQ,
   (BYTE *) &curdbid, sizeof (dbid_t));
 startscan(db_sdes, NC1_DATABASE, SCAN_NORMAL);
 dbbuf = getnext(db_sdes);
 bufpredirty(dbbuf); /* sync change to page */
 /* change status in Sysdatabases row */
 datstat =
    GETSHORT(&((DATABASE *)(db_sdes->srow))->datstat);
datstat &= ~DBT_NOTREC;
MOVE_FIXED(&datstat, &((DATABASE *) db_sdes->srow)->
 datstat, sizeof (datstat));
/* change dbt3 status field in the Sysdatabases row */
stat3p = (BYTE *) collocate(db_sdes->srow,
      Sysdatabases[DAT_STATUS3].scoloffset,
      IND_GET_01ROW_MINLEN(db_sdes->sdesp),
      IND_GET_01ROW_MAXLEN(db_sdes->sdesp),
      SDES_ROWFORMAT(db_sdes), &stat3len);
if (stat3p != NULL)
{
 MEMMOVE(stat3p, &status3, sizeof(int32));
 status3 &= ~DBT3_QUIESCE_DB_BITS;
 if (status3 & DBT3_DBSHUTDOWN_FROM_ACCESS)
 {
  /*
  ** If DBT3_DBSHUTDOWN_FROM_ACCESS is
  ** set and we are in recovery there
  ** will be no users of the database
  ** so it is as good as the database
  ** having been shut down successfully.
  */
  status3 |= DBT3_DBSHUTDOWN_COMPLETED;
  dbt_updstatus(dbt, DBT3_DBSHUTDOWN_COMPLETED,
        DBT_STATUS3, BIT_SET);
 }
 ha_state = ha_get_server_state();
 if (HA__FAILEDOVERSTATE(ha_state))
 {
  if ((status3 & DBT3_HA_PROXYDB) &&
      (!(status3 & DBT3_FAILEDOVER_DATABASE)))
  {
   status3 |=
    (DBT3_DBSHUTDOWN_FROM_ACCESS |
     DBT3_DBSHUTDOWN_COMPLETED);
   dbt_updstatus(dbt,
    DBT3_DBSHUTDOWN_FROM_ACCESS,
    DBT_STATUS3, BIT_SET);
   dbt_updstatus(dbt,
    DBT3_DBSHUTDOWN_COMPLETED,
    DBT_STATUS3, BIT_SET);
  }
 }
 MEMMOVE(&status3, stat3p, sizeof(int32));
}
/*
** When server is in failed-over state, the local
** proxy database are renamed with a prefix as
** defined in HA_LOCAL_PROXY_PREFIX_STR.
** So, if we are recovering such a database, then
** disallow external access to such a database.
*/
if (SQLSTR_EQ(dbt->dbt_dbname,
       strlen(HA_LOCAL_PROXY_PREFIX_STR),
       (BYTE *) HA_LOCAL_PROXY_PREFIX_STR,
       strlen(HA_LOCAL_PROXY_PREFIX_STR)))
{
 MEMMOVE(stat3p, &status3, sizeof(int32));
 SYB_ASSERT((status3) & DBT3_HA_PROXYDB);
 status3 |= DBT3_DBSHUTDOWN_FROM_EXT_ACCESS;
 MEMMOVE(&status3, stat3p, sizeof(int32));
 /*
 ** Reset the status3 bit in dbtable indicate that
 ** external access is disallowed.
 */
 dbt_updstatus(dbt, DBT3_DBSHUTDOWN_FROM_EXT_ACCESS,
   DBT_STATUS3, BIT_SET);
}
/* Make sure the row hits the page */
REPLACE_ROW_ON_PAGE(db_sdes, dbbuf);
/*
** We are in recovery and this is a non-transactional
** update update so use bufdirty_nx().
*/
bufdirty_nx(dbbuf, db_sdes);
/*
** TESTING POINT:
** If trace flag 3466 is on, raise a fatal error here
** to simulate hardware errors that could happen within
** bufwrite().
*/
if (TRACECMDLINE(RECOVER, 66))
{
 ex_raise(RECOVER, REC_RETURN, EX_HARDWARE, 3);
}
bufwrite(dbbuf, db_sdes);
endscan(db_sdes);
backout_ctx->db_sdes = NULL;
closetable(db_sdes);
dbt_notrecoff(dbt);
/*
** This bit was cleared above in Sysdatabases.
** Clear it as well in the dbatble.
*/
dbt_updstatus(dbt, DBT3_QUIESCE_DB_BITS,
  DBT_STATUS3, BIT_RESET);
/* OMNI support */
dso_text_recover(dbt);
/*
```

```
  ** Bring the database online.
  ** - the second parameter indicates that we should
  **   bring it online, unless it is in a state where
  **   it shouldn't be online (e.g. a load sequence).
  ** - the third parameter indicates whether we are
  **   bring this online for normal or standby access.
  **   If the database was brought online for standby
  **   access previously, we should do the same now.
  */
  onl_online_db(dbt, FALSE, ((dbt->dbt_stat2 & DBT2_ONL_STANDBY) ?
    ONL_STANDBYMODE : ONL_NORMAL));
  /*
  ** At the end of recovery, when it is safe to write
  ** log records, decrement the pss count and release
  ** the address lock on the coordinating transaction
  ** It is important to clean up the xdes before the
  ** lock on it released so that it is not up for
  ** grabs.
  */
  if (pss->prectable)
  {
   rec_release_rectable(pss->prectable);
  }
 }
 backout_ctx->backout_dbt = (DBTABLE *) NULL;
 closedb(USEPREV);
 return;
}
/*
** REC_CLEANUP_RECOVERY_ITEM
**
** Purpose: This procedure is called after the recovery work for a
**  database is done. If this is called during normal_cleanup,
**  it means the database is recovered and onlined. Otherwise,
**  it is called during exception handling.
**
**  It will clean up the rec_order_item entry for this database
**  in Resource.
**
** Parameters:
** rec_order_num - the rec_order_num for the target database
** normal_cleanup - TRUE if this function is called during normal
**    clean up of a recovery item.
**    FALSE if this function is called as part of
**    exception handling.
**
** Return:
** None.
**
** Side Effects:
```

```
** 1. The status bits regarding recovery state for the item are cleared;
** 2. set REC_ITEM_RECOVERED or REC_ITEM_FAILED in the recovery item
**    depending on if the function is called during normal cleanup
**    or during exception handling;
** 3. the number of databases done will be incremented;
** 4. If this item is the first strict order item, the first strict order
**    item in rec_order_info will be reset to the next strict order item
**    or 0 if there isn't any.
**
** Synchronization:
** The whole function is done under spinlock Resource->ha_spin.
**
** History:
**  4/30/02 (fzhou) - written.
*/
void
rec_cleanup_recovery_item(int rec_order_num, SYB_BOOLEAN normal_cleanup)
{
 REC_ORDER_INFO *rec_order_info;
 REC_ORDER_ITEM *passed_in_item;
 int  num_items;
 int  *first_strict_order;
 SYB_BOOLEAN onlined_with_strict_order;
 LOCALPSS(pss);
 /* Initialize local variables. */
 num_items = 0;
 onlined_with_strict_order = FALSE;
 rec_order_info = Resource->rrecovery_info.rec_order_info;
 /*
 ** If the global structure doesn't exist, no need to go further.
 ** This could happen if we are called during cleanup when the
 ** pss is terminated.
 */
 if (!rec_order_info)
 {
  return;
 }
 /*
 ** Get the total number of recovery items in the array.
 ** No need to do it under spinlock because this field is not modified
 ** after the rec_order_info is populated before any recovery thread
 ** is spawned.
 */
 num_items = rec_order_info->rec_order_size;
 /*
 ** First check and make sure that passed in rec_order_num is
 ** valid. Otherwise, return.
 */
 if ((rec_order_num > num_items) || (rec_order_num < 1))
 {
```

```
 SYB_ASSERT(0);
 return;
}
passed_in_item =  &rec_order_info->rec_order_item[rec_order_num];
first_strict_order = &rec_order_info->first_offline_strict_order_item;
/*
** This is only for internal debugging, so it is ok to get the values
** without spinlock.
*/
if (TRACECMDLINE(RECOVER, 56))
{
 scerrlog("The first strict order item is %d. The passed in item %d has status of %d\n",
    *first_strict_order, rec_order_num,
    (int)passed_in_item->status);
}
P_SPINLOCK(Resource->ha_spin);
/*
** If any of the following is true, do not clean up the item.
**
** 1. If the database represented by this item was put to a delayed
**    online mode by having the REC_ITEM_WAITING_TO_ONLINE bit set
**    AND this function is called during normal_cleanup.
**    This thread cannot clean up this item because there is still
**    work left to do. Another thread will pick up this item later
**    and finish the work.
**    In this case, assert that the pss registered in
**    passed_in_item->recovery_pss _is_ the current pss.
** OR
** 2. If the pss registered in the passed_in_item which is
**    recovering this database is no longer this pss. This could
**    happen if some other thread has picked up this
**    WAITING_TO_ONLINE item and started working on it before this
**    thread comes here to clean up the item. The other thread
**    would have cleared the REC_ITEM_WAITING_TO_ONLINE bit and
**    register their pss in the recovery_pss in the item.
**    This thread cannot clean up this item, because the item
**    now belongs to another recovery thread.
**    In this case, assert that REC_ITEM_WAITING_TO_ONLINE is not
**    set in the item.
*/
if ( ((passed_in_item->status & REC_ITEM_WAITING_TO_ONLINE) &&
 (normal_cleanup)) ||
     (passed_in_item->recovery_pss != pss) )
{
 /*
 ** Assert the two scenarios where we could end up not cleaning
 ** up the recovery item.
 */
 SYB_ASSERT( ((passed_in_item->recovery_pss == pss) &&
       (normal_cleanup) &&
        (passed_in_item->status &
     REC_ITEM_WAITING_TO_ONLINE))  ||
         ((passed_in_item->recovery_pss != pss) &&
        (!(passed_in_item->status &
     REC_ITEM_WAITING_TO_ONLINE))) );
  V_SPINLOCK(Resource->ha_spin);
  if (TRACECMDLINE(RECOVER, 56))
  {
   scerrlog("The database was not onlined due to some online ordering conflict. Therefore, do not clean up the
information in the recovery item %d.\n",
     rec_order_num);
  }
  return;
 }
 /*
 ** Assert that the item is not already cleaned up.
 ** The only non-error condition where the item could have been
 ** cleaned up before this thread comes here is:
 ** T1 (this thread), T2 (another recovery thread)
 ** following events happened in sequence:
 ** -- T1 set REC_ITEM_WAITING_TO_ONLINE in the item;
 ** -- T2 finishes its previous database (which was reason why this
 **    db couldn't be onlined right away), call
 **    rec_getnextdb_to_recover() to pick up the next database to
 **    recover;
 ** -- T2 picks up this database:
 **    Clear REC_ITEM_WAITING_TO_ONLINE bit;
 **    Set REC_ITEM_TO_BE_ONLINED bit;
 **    Set recovery_pss in the item to the pss of T2;
 ** -- T2 calls rec_makedb_accessible() to online the database;
 ** -- T2 calls rec_cleanup_recovery_item() to clean up the item;
 ** -- T2 cleaned up the item (REC_ITEM_RECOVERED is set);
 ** -- T1 calls rec_cleanup_recovery_item() and finds that the
 **    item is already marked REC_ITEM_DONE.
 **
 ** this scenario would've been taken care of by the check at
 ** the beginning of this function, because the
 ** recovery_pss in the item would have been reassigned with
 ** the pss of T2, and therefore T1 would've returned and shouldn't
 ** get to this assertion check.
 */
 SYB_ASSERT (!(passed_in_item->status & REC_ITEM_DONE));
 /*
 ** If this item was onlined with strict order, remember it before
 ** we clear the status bit.
 */
 if (passed_in_item->status & REC_ITEM_ONL_WITH_STRICT_ORDER)
 {
  onlined_with_strict_order = TRUE;
 }
```

```
/*
** Clean up all status bits related to item's recovery state
** because it is not in recovery any more.
*/
passed_in_item->status  &= ~(REC_ITEM_RECOVERY_STATE);
/*
** Set the end state for the item. If we are called during
** normal clean up after a successful recovery, mark the
** item RECOVERED.
*/
if (normal_cleanup)
{
 passed_in_item->status |= REC_ITEM_RECOVERED;
}
else
{
 /* Otherwise, mark the item REC_ITEM_FAILED. */
 passed_in_item->status  |= REC_ITEM_FAILED;
}
/* Increment the book keeping field in the global structure. */
rec_order_info->num_dbs_done ++;
/*
** If this item had a strict recovery order, and it is the
** first strict online order item, set the first strict order
** item to the next item with strict recovery order.
**
** It is possible that this passed in item is not the first
** strict order item because this function can be called for
** cleanup during exception handling when the recovery of
** the item is interrupted by exception and therefore didn't
** go through the same checking route for recovery order in
** onl_online_db().
*/
if ((onlined_with_strict_order) &&
 (*first_strict_order == rec_order_num))
{
 /*
 ** if this is the first strict order item, we need to
 ** do two things:
 ** 1. find the next item with strict online order,
 **    if there is any, and point
 **    first_offline_strict_order_item to it.
 ** 2. For all items in between that are not done yet,
 **    set the REC_ITEM_ONL_IMMEDIATELY bit because
 **    they no longer need to be checked against this
 **    passed in recovery item's recovery state when
 **    they are about to be made online.
 */
 rec__set_next_strict_order_item(rec_order_num,
     num_items, TRUE);
```

```
  V_SPINLOCK(Resource->ha_spin);
  if (TRACECMDLINE(RECOVER, 56))
  {
   scerrlog("The new first_strict_order is %d.\n",
       *first_strict_order);
  }
  return;
 }
 V_SPINLOCK(Resource->ha_spin);
 return;
}
/*
** REC_MAKEDB_ACCESSIBLE
** This function will clear the DBT2_OFFLINE and DBT2_AUTO_ONL bits in
** the current database pointed to by the curdbid in the context structure,
** and flush the status to disk. Thus, the database is made accessible
** to clients. An online message will also be printed to indicate that
** the database is online.
**
** Parameter:
** None.
**
** Returns:
** None.
**
** History:
** 3/2003 - created (fzhou)
*/
void
rec_makedb_accessible(void)
{
 SDES    *db_sdes;
 BUF   *dbbuf;
 REC_ORDER_INFO  *rec_order_info;
 REC_CALLER_BKOUT_CTX *ctx;
 SYB_BOOLEAN  boot_recovery;
 int   onlmsg;
 dbid_t   metadbid;
 dbid_t   curdbid;
 int   rec_order_num;
 BYTE   *namep;
 int   dbnlen;
 char   dbname[MAXNAME];
 SARG   idsarg;
 rec_order_info = Resource->rrecovery_info.rec_order_info;
 ctx = (REC_CALLER_BKOUT_CTX *)REC_CALLER_CTX_FROM_PSS(Pss);
 SYB_ASSERT(ctx != (REC_CALLER_BKOUT_CTX *)NULL);
 /*
 ** Get the necessary information from the context structure because
 ** they were set in the caller before we come here.
```

```
*/
curdbid = ctx->curdbid;
rec_order_num = ctx->rec_order_num;
/* Get the meta database id to open sysdatabases. */
metadbid = rec_order_info->metadbid;
boot_recovery = (metadbid == MASTERDBID);
/*
** Before we do any work, assert that the backout_dbt in the context
** structure is NULL.
*/
SYB_ASSERT(ctx->backout_dbt == (DBTABLE *)NULL);
/*
** Open sysdatabases in the meta database, which is master database
** in boot recovery, and master_companion in failover recovery.
*/
if (boot_recovery)
{
 db_sdes = OPEN_SYSTEM_TABLE((objid_t) SYSDATABASES, metadbid,
     Resource->rmasterdbt);
}
else
{
 db_sdes = OPEN_USER_TABLE((objid_t) SYSDATABASES, metadbid, 1,
    (BYTE *)"sysdatabases", 12);
}
if (db_sdes == (SDES *)NULL)
{
 SYB_ASSERT(0);
 return;
}
/* store the sdes in backout context structure. */
ctx->db_sdes = db_sdes;
/*
** Set up scan for sysdatabases.
*/
db_sdes->sstat |= (SS_FGLOCK | SS_L1LOCK);
initarg(db_sdes, &idsarg, 1);
setarg(db_sdes, &Sysdatabases[DAT_DBID], EQ,
 (BYTE *) &curdbid, sizeof (dbid_t));
/* get the next database row */
startscan(db_sdes, NC1_DATABASE, SCAN_FIRST);
if ((dbbuf = getnext(db_sdes)) == NULL)
{
 CLOSE_SDES(&ctx->db_sdes);
 return;
}
/*
** Update the DBT2_OFFLINE bit in dbtable and sysdatabases.
*/
dbt_upd_offlbits((DBTABLE *)NULL, db_sdes, dbbuf, CLEAR_DBOFFLINE);
```

```
 /* Get the database name for print. */
 namep = collocate(db_sdes->srow,
    Sysdatabases[DAT_NAME].scoloffset,
    OFFSETOF(DATABASE, datlen),
    Sysdatabases[DAT_NAME].scollen,
    SDES_ROWFORMAT(db_sdes), &dbnlen);
 MEMMOVE(namep, dbname, dbnlen);
 CLOSE_SDES(&ctx->db_sdes);
 /*
 ** Now that the database is really considered online, print the
 ** online message and clean up the recovery item for this database.
 */
 onlmsg = rec_order_info->rec_order_item[rec_order_num].onlmsg;
 ex_callprint(onlmsg, EX_INFO, 2, dbnlen, dbname);
 return;
}
/*
** REC_GETNEXTDB_TO_RECOVER
**
** Type: internal function (called from recover_by_order)
**
** Purpose:  This procedure finds the next database that needs to be
**  recovered. It searches the rec_order_item array in
**  Resource->rrecovery_info.rec_order_info under spinlock.
**
**  The search through the list of recovery items will always
**  start from item 1 because of the following two reasons:
**  1. During tuning period, we will skip user created tempdbs
**     because their recovery path are different than normal
**     databases and is not suitable for sampling. Thus, once
**     the tuning has completed, we will need to pick up the
**     skipped user created tempdbs.
**  2. When a database could not be onlined right away due to
**     online order conflict, it will be put to a deferred online
**     mode by having REC_ITEM_WAITING_TO_ONLINE bit set in its
**     corresponding recovery item. These items must be picked up
**     and made online as soon as they can be made online and in
**     order of their recovery item numbers.
**
** Parameters:
**  rec_caller_ctx - The context structure, in which the curdbid
**    and rec_order_num fields will be filled with
**    the information of the next db to recover,
**    if there is any.
**
** Returns:
**  REC_GOT_NOMORE_DB - no more db to recover
**  REC_GOT_NEXT_DB  - successfully found next db to recover
**  REC_GOT_TEMPDB  - successfully found a user created
**       tempdb.
**  REC_ONLINE_DB  - successfully found a database which
**       was waiting to online. Will online
**       the database.
**  REC_NEED_TO_EXIT - user has decided to go back to
**     serial recovery, so the recovery
**     thread will exit.
**  REC_WAKEUP_FROZEN_THREAD
**     - the server is running too many
**     threads than it can handle, therefore
**     this thread needs to exit to reduce
**     the number of running threads.
**     Before exiting, the caller to this
**     func will wakeup any thread that
**     has been sleeping because the server
**     was running with too many threads.
**
** Synchronization:
**  If this function is called during parallel recovery, the
**  ha_spin will be obtained before entering this function.
**  Otherwise, there is no need for spinlock protection.
**
** History:
**  4/30/02 (fzhou) - written
*/
int
rec_getnextdb_to_recover(REC_CALLER_BKOUT_CTX *rec_caller_ctx)
{
 LOCALPSS(pss);
 int  num_of_items;
 int  cnt;
 int  first_strict_item_num;
 SYB_BOOLEAN need_to_skip_tempdb;
 SYB_BOOLEAN tempdb_skipped;
 REC_ORDER_INFO *rec_order_info;
 REC_ORDER_ITEM *temp_rec_item;
 RECOVERY_INFO *recovery_info;
 SYB_BOOLEAN saw_not_done_item;
 int  search_cnt;
 recovery_info  = &Resource->rrecovery_info;
 rec_order_info = recovery_info->rec_order_info;
 SYB_ASSERT(rec_order_info != (REC_ORDER_INFO *)NULL);
 first_strict_item_num = rec_order_info->first_offline_strict_order_item;
 num_of_items = rec_order_info->rec_order_size;
 need_to_skip_tempdb = FALSE;
 search_cnt = 0;
 /*
 ** If the server is having too many threads than it can handle,
 ** we need to exit to reduce the number of running threads. Before
 ** the thread exits, wake up any thread that is frozen because
 ** there are too many threads running.
```

```
*/
if (recovery_info->status & REC_INFO_TOO_MANY_THREADS)
{
 /*
 ** Assert that the ha_spin lock was obtained by the caller,
 ** as we could not have had this status set if it were not
 ** during parallel recovery. And for parallel recovery,
 ** we have to have spinlock protection.
 */
 SPINLOCKHELD(Resource->ha_spin);
 SYB_ASSERT(recovery_info->status & REC_INFO_PARALLEL);
 /*
 ** With this value returned, the caller will wakeup
 ** thread that has been sleeping because of too many threads
 ** were running on the server.
 */
 return REC_WAKEUP_FROZEN_THREAD;
}
/*
** If the user has decided to go back to serial recovery, and
** changed the configuration parameter value to 1, all recovery
** threads will exit after finishing up with the current item.
*/
if (recovery_info->status & REC_INFO_GOBACK_TO_SERIAL)
{
 /*
 ** If this is a recovery thread, return to caller indicating
 ** that this thread should exit, because user has requested
 ** to go back to serial recovery and this way, the recovery
 ** threads will be drained out.
 */
 if (pss->pprocess_type == RECOVERY)
 {
  return REC_NEED_TO_EXIT;
 }
 /*
 ** otherwise, this must be the initial thread doing serial
 ** recovery already, clear the bit and fall through.
 */
 else
 {
  recovery_info->status &= ~(REC_INFO_GOBACK_TO_SERIAL);
 }
}
if (recovery_info->status & REC_INFO_FAILOVER_FAIL)
{
 return REC_NEED_TO_EXIT;
}
/*
** If the server is not having problem handling threads, nor was it
** given command to go back to serial recovery, continue recovering
** by getting the next item that hasn't been picked up.
*/
/*
** If server is still in tuning process, skip user created tempdb.
*/
if (!(recovery_info->status & REC_INFO_TUNE_COMPLETE))
{
 need_to_skip_tempdb = TRUE;
}
search_again:
/*
** Initialize variables which need to be refreshed before we
** start the search.
*/
tempdb_skipped = FALSE;
saw_not_done_item = FALSE;
/*
** Increment the search cnt for each search. This count is used
** for sanity checking to make sure that we don't go into infinite
** search.
*/
search_cnt ++;
/*
** The maximum number of searches that we can do is 2, because we
** could have skipped some tempdbs in our first search, but found
** no other user dbs, and had to search again to pick up the tempdbs.
*/
SYB_ASSERT(search_cnt <= 2);
for (cnt = 1; cnt <= num_of_items; cnt++)
{
 temp_rec_item = &rec_order_info->rec_order_item[cnt];
 if (temp_rec_item->status & REC_ITEM_NOT_RECOVERED)
 {
  if (need_to_skip_tempdb &&
     (temp_rec_item->status & REC_ITEM_USER_TEMPDB))
  {
   tempdb_skipped = TRUE;
   continue;
  }
  temp_rec_item->status &= ~(REC_ITEM_NOT_RECOVERED);
  temp_rec_item->status |= REC_ITEM_RECOVERING;
  rec_caller_ctx->curdbid = temp_rec_item->dbid;
  rec_caller_ctx->rec_order_num = cnt;
  /*
  ** Store the pointer to this pss in rec_order_item.
  ** This is used to identify the pss of the thread
  ** that is working on this item.
  */
  temp_rec_item->recovery_pss = pss;
 /*
 ** If this is a user created tempdb, return
 ** to indicate that we will recover a user tempdb.
 */
 if (temp_rec_item->status & REC_ITEM_USER_TEMPDB)
 {
  return REC_GOT_TEMPDB;
 }
 else
 {
  return REC_GOT_NEXT_DB;
 }
}
else if (temp_rec_item->status & REC_ITEM_WAITING_TO_ONLINE)
{
 /*
 ** If the database is waiting to be onlined, test
 ** to see if it can be made online now.
 ** The same check used in rec_online_order_conflict()
 ** is used here: If any of the following is true,
 ** the database can be onlined right away:
 **
 ** 1. there is no strict online order item (i.e.
 **    first_strict_item_num == 0); OR
 ** 2. the item number of this database is less
 **    than the first_strict_item_num; OR
 ** 3. this item is the first strict order item, and
 **    there is no "not done" item _before_ this item,
 **    not including this item.
 */
 if ( (first_strict_item_num == 0) ||
      (cnt < first_strict_item_num) ||
      ((cnt == first_strict_item_num) &&
       !saw_not_done_item) )
 {
  temp_rec_item->status &=
    ~(REC_ITEM_WAITING_TO_ONLINE);
  temp_rec_item->status |=
    REC_ITEM_TO_BE_ONLINED;
  rec_caller_ctx->curdbid = temp_rec_item->dbid;
  rec_caller_ctx->rec_order_num = cnt;
  /*
  ** Store the pointer to this pss in
  ** rec_order_item.
  ** This is used to identify the pss of the
  ** thread that is working on this item.
  */
  temp_rec_item->recovery_pss = pss;
  return REC_ONLINE_DB;
 }
 }
 /*
 ** Keep a record if there is a "not done" item _before_
 ** the current item.
 ** This is used to determine if a particular item waiting
 ** to online can be brought online.
 ** This has to be done after the check for the
 ** REC_ITEM_WAITING_TO_ONLINE bit because the current item
 ** rec_order_info->rec_order_item[cnt] will always be
 ** !REC_ITEM_DONE.
 */
 if (!saw_not_done_item &&
     !(temp_rec_item->status & REC_ITEM_DONE))
 {
  saw_not_done_item = TRUE;
 }
} /* end of for loop */
/*
** If we didn't find any item to recover, check to see if we have
** skipped any tempdb because of tuning, if so, go back to pick up
** those tempdbs.
*/
if (tempdb_skipped)
{
 need_to_skip_tempdb = FALSE;
 goto search_again;
}
else
{
 /* Assert that we have walked through all items in the list. */
 SYB_ASSERT(cnt == (num_of_items + 1));
 /* Otherwise, return REC_GOT_NOMORE_DB. */
 return REC_GOT_NOMORE_DB;
}
}
/*
** REC__COLLECT_STATISTICS
**
** Purpose:
** This is the function tuning thread calls to collect statistics.
**  For each sample, it will:
**  1. pause for a predefined sampling period.
** 2. Then fetch the global sampling counter, and if necessary
**    go for the next sample.
** After all samples are collected, the average will be computed
** based on the sample counters, and the result will be returned to
** the caller.
**
**  If the recovery has completed during this period, or the statistics is
** marked as invalid, return corresponding return values to the caller.
```

```
**
** Parameters:
** stat_index - The index to the array of statistics collected for
**       each thread.
**
** Returns:
** the new statistics OR
** REC_RECOVERY_COMPLETE - recovery has already completed before we
**    finish collecting this statistics.
**
** REC_INVALID_STAT - this statistics is invalid.
**    This is returned when the latest spawned
**     thread completed the analysis pass before
**     sample period ended.
**
** History:
** 6 2002 - (fzhou) created
*/
SYB_STATIC int
rec__collect_statistics(int stat_index)
{
 LOCALPSS(pss);
 int  last_sample_counter;
 int  retvalue;
 int  sample_time;
 int  first_sample;
 RECOVERY_INFO *recovery_info;
 REC_ORDER_INFO *rec_order_info;
 REC_ORDER_ITEM *cur_rec_item;
 /* initialize the return value to 0 before the sampling starts */
 retvalue = 0;
 /* Initialize local pointers to the global structure */
 recovery_info = &Resource->rrecovery_info;
 rec_order_info = recovery_info->rec_order_info;
 cur_rec_item = &rec_order_info->rec_order_item[stat_index];
 /*
 ** If trace flag 3461 is on, do not throw away the first sample,
 ** and therefore the loop starts at 1. Otherwise, loop starts
 ** at 0, because later samples whose (sample_time == 0) will
 ** not be counted toward the new statistics.
 */
 if (TRACECMDLINE(RECOVER, 61))
 {
  sample_time = 1;
 }
 else
 {
  sample_time = 0;
 }
 /*
** Acquire the spinlock before we access the global structure.
*/
P_SPINLOCK(Resource->ha_spin);
/*
** This bit shouldn't be set, but clear it anyway before we
** start a new statistics collection
*/
recovery_info->status &= ~REC_INFO_INVALID_STAT;
for(; sample_time <= TOTAL_NUM_SAMPLES; sample_time ++)
{
 /*
 ** Before we start to collect another sample,
 ** 1. initialize the global counter to 0;
 ** 2. set the status bit to inform buffer manager to collect
 ** statistics.
 */
 rec_order_info->read_counter = 0;
 recovery_info->status |= REC_INFO_COLLECT_STAT;
 V_SPINLOCK(Resource->ha_spin);
 /*
 ** Pause this thread for the sample period. The attention flag
 ** is set to FALSE because we don't want to be woken up by
 ** attention.
 **
 ** Since this is not in a performance sensitive code path,
 ** no need for a mda identifier. Use UNUSED instead.
 */
 (void) uppause(1 * USECS_PER_SEC, (syb_event_t *) NULL, FALSE, UNUSED);
 /*
 ** When we are woken up one of the following situations
 ** are possible
 ** a) Recovery has completed
 ** b) A statistic sample was found to be invalid
 ** c) A valid statistic was obtained
 */
 P_SPINLOCK(Resource->ha_spin);
 /* After the sample period, stop the statistics collection */
 recovery_info->status &= ~REC_INFO_COLLECT_STAT;
 /*
 ** If this is the to-be-thrown away sample, get the
 ** counter value so that it can be printed out after
 ** the spinlock is released.
 */
 if (sample_time == 0)
 {
  first_sample = rec_order_info->read_counter;
 }
 /*
 ** Get the global counter for this sample, and stored
 ** it in the global structure.
 */
 else
 {
  cur_rec_item->sample_result[sample_time - 1]
    = rec_order_info->read_counter;
  /*
  ** Add this sample counter to the total statistics which
  ** will be used to calculate the average counter.
  */
  retvalue += rec_order_info->read_counter;
 }
 /*
 ** If recovery has already completed, set the return value and
 ** break out of the loop to do some clean up.
 */
 if (!(recovery_info->status & REC_INFO_PARALLEL))
 {
  retvalue = REC_RECOVERY_COMPLETE;
  break;
 }
 /*
 ** If this was found to be an invalid statistics,
 ** 1. Store the this sample counter to a local variable to
 ** print it later after we release the spinlock.
 ** 2. Set the this sample counter to -1 in the global
 ** structure indicating an invalid statistics.
 ** 3. Set the return value and break out of the loop to
 ** do some clean up.
 */
 if (recovery_info->status & REC_INFO_INVALID_STAT)
 {
  last_sample_counter = rec_order_info->read_counter;
  if (sample_time == 0)
  {
   cur_rec_item->sample_result[sample_time] = -1;
  }
  else
  {
   cur_rec_item->sample_result[sample_time-1] = -1;
  }
  retvalue = REC_INVALID_STAT;
  break;
 }
}
/*
** The statistics collection is done for this thread,
** 1. clear the field in the global structure which is used to
** determine invalid sampling during statistics collection;
** 2. clear the REC_INFO_INVALID_STAT bit.
*/
```

```
 rec_order_info->latest_spawned_thread_kpid = 0;
 recovery_info->status &= ~REC_INFO_INVALID_STAT;
 V_SPINLOCK(Resource->ha_spin);
 /* calcualte the average sample counter. */
 if ((retvalue != REC_RECOVERY_COMPLETE) &&
    (retvalue != REC_INVALID_STAT))
 {
  retvalue = retvalue / TOTAL_NUM_SAMPLES;
 }
 /*
 ** If trace flag 3456 is on, print some diagnostic information
 ** according to different ret values.
 */
 if(TRACECMDLINE(RECOVER, 56))
 {
  if (!TRACECMDLINE(RECOVER, 61))
  {
   scerrlog(" The first sample was not used. Its value is %d.\n",
    first_sample);
  }
  if (retvalue == REC_RECOVERY_COMPLETE)
  {
   scerrlog("server is no longer in parallel recovery. Stop collecting statistics.\n");
  }
  else if (retvalue == REC_INVALID_STAT)
  {
   scerrlog("The statistics is not valid. We have collected %d samples for it. The last sample value is %d. \n",
    (sample_time + 1), last_sample_counter);
  }
  else
  {
   scerrlog("We have collected a valid statistics. The values is %d.\n",
    retvalue);
  }
 }
 return retvalue;
}
/*
** REC__STAT_DEGRADED
**
** Purpose:
** This is the function tuning thread calls to determine if the
** statistics collected has showed unacceptable degradation comparing
** to the previous statistics.
**
** It will call rec__collect_statistics() to collect the new statistics
** and then compare the new stat with the passed in prev stat to
** determine the result.
**
**  If the recovery has completed or rec__collect_statistics() got an
** invalid statistics, return corresponding return values to the caller.
**
** Caller will take approporiate actions according to the return results.
**
** Parameters:
** stat_index - (IN) The index to the array of statistics collected for
**   each thread.
** prev_stat - (IN/OUT) This is a pointer to the previous statistics.
**   The value will be used to compare the new statistics
**   with. If the result is that the new statistics doesn't
**   show unacceptable degradation, this field will be filled
**   with the new statistics, which will be used the next
**   time this function is called.
**
** Returns:
** REC_RECOVERY_COMPLETE - Recovery has already completed.
** REC_INVALID_STAT - Statistics collected is invalid.
** REC_STAT_DEGRADATION - New statistics has showed unacceptable
**    degradation over the previous statistics.
** REC_STAT_NO_DEGRADATION - New statistics has NOT showed unacceptable
**    degradation over the previous statistics.
**
** History:
** 2/2003 -  (fzhou) created
*/
SYB_STATIC int
rec__stat_degraded(int stat_index, int *prev_stat)
{
 int  retvalue;
 int  new_stat;
 int  old_stat;
 double  diff_percent;
 /* collect the new statistics */
 retvalue = rec__collect_statistics(stat_index);
 /*
 ** If the recovery has completed while collecting statistics or
 ** the current statistics is invalid, return to caller.
 */
 if ((retvalue == REC_RECOVERY_COMPLETE) ||
    (retvalue == REC_INVALID_STAT))
 {
  return (retvalue);
 }
 /*
 ** We have collected a valid new statistics, store it in
 ** the new_stat.
 */
 new_stat = retvalue;
 old_stat = *prev_stat;
 /*
 ** Compare the new statistics with the value of the passed in pervious
 ** statistics to see if there is any unacceptable degradation.
 ** If trace flag 3460 is on, always simulate performance degradation
 ** on the second thread.
 */
 if ( ((TRACECMDLINE(RECOVER, 60)) && (stat_index == 2))  ||
     (new_stat < ((1 - MAX_PERFORMANCE_DROP_ALLOWED) * old_stat)) )
 {
  if (TRACECMDLINE(RECOVER, 60))
  {
   diff_percent = (new_stat - old_stat) * 100.0 /
      (old_stat * 1.0);
   scerrlog("Simulate performace degradation. The actual performance change is %.2f%% (positive number means
performance numbers increased and negative number represents performance degradation).\n",
     diff_percent);
  }
  retvalue = REC_STAT_DEGRADATION;
 }
 else
 {
  retvalue = REC_STAT_NO_DEGRADATION;
 }
 /*
 ** After this statistics has been used and compared, it will be
 ** used as the previous statistics in the next run of statistics
 ** collection and analysis.
 */
 *prev_stat = new_stat;
 return (retvalue);
}
/*
** REC__FREEZE_RECOVERY_THREAD
**
** Purpose:
** This procedure searches through the rec_order_info and freeze one
** recovery thread if needed. This is done as part of the tuning process.
**
** If there is still need to freeze thread after we acquire the spinlock:
** 1. Search the list of recovery items backwards to find the first
**    item that is marked as REC_ITEM_RECOVERING, and get the pss of
**    the recovery thread which is recovering this database.
** 2. set the REC_INFO_TOO_MANY_THREADS bit in Resource.
**    This status bit will make the next recovery thread which tries
**    to pick up an item exit, and therefore reducing the running
**    recovery threads.
** 3. If we have found such a pss, set PEXT_FREEZE_RECOVERY_THREAD in the
**    pss.
**    When the recovery thread checks for this bit and finds it set,
**    it will call rec_freeze_thread() to go to sleep on the
**    REC_INFO_TOO_MANY_THREADS bit and therefore be frozen.
```

```
**
** Parameters:
** None.
**
** Returns:
** nothing.
**
** Side Effects:
** The following two effects are not always happening, see the above
** function purpose description for detail:
** * REC_INFO_TOO_MANY_THREADS is set in recovery_info.status
** * one of the recovery threads has the PEXT_FREEZE_RECOVERY_THREAD set
**    in the pss
**
** Callers:
** rec_run_parallel_recovery
**
** Synchronization:
** The whole function is protected under spinlock.
**
** History:
** 4/02 (fzhou) - created
*/
SYB_STATIC void
rec__freeze_recovery_thread()
{
 int  curnum_running_threads;
 int  optimal_num_threads;
 int  count;
 spid_t  frozen_pss = 0;
 PSS      *recovery_pss = NULL;
 REC_ORDER_INFO *rec_order_info;
 RECOVERY_INFO *recovery_info;
 recovery_info = &Resource->rrecovery_info;
 rec_order_info = recovery_info->rec_order_info;
 P_SPINLOCK(Resource->ha_spin);
 curnum_running_threads = rec_order_info->num_rec_threads;
 optimal_num_threads = rec_order_info->optimal_num_rec_threads;
 /*
 ** Do not freeze any recovery thread if either of the following
 ** cases is true:
 ** 1. the number of recovery threads is 1 or 0, OR
 ** 2. the number of recovery threads is no more than the
 ** optimal number of recovery threads determined by the caller.
 **
 ** These cases could happen because it is possible that some
 ** recovery threads have completed and exited during our last
 ** sampling period, OR tuning thread didn't find a need to freeze
 ** a thread.
 */
```

```
if ((curnum_running_threads <= 1) ||
    (curnum_running_threads <= optimal_num_threads))
{
 V_SPINLOCK(Resource->ha_spin);
 return;
}
/*
** Assert that the optimal number of recovery threads is 1 less
** than the number of current running recovery threads before
** we start the freeze process.
*/
SYB_ASSERT(optimal_num_threads == (curnum_running_threads-1));
/*
** Search the list of recovery items backwards for
** the first item that is being recovered.
**
** In another word, we are freezing the recovery thread
** in the opposite sequence as the recovery order
** sequence.
*/
for (count = rec_order_info->rec_order_size; count >= 1; count--)
{
 if (rec_order_info->rec_order_item[count].status &
      REC_ITEM_RECOVERING)
 {
  /*
  ** found the item.
  ** get the pss and Break out of the loop.
  */
  recovery_pss =
      rec_order_info->rec_order_item[count].recovery_pss;
  break;
 }
}
/*
** Set the status to indicate that the server is running more threads
** than it can handle.
*/
recovery_info->status |= REC_INFO_TOO_MANY_THREADS;
/* If we have found a recovery pss to freeze, set the FREEZE bit. */
if (recovery_pss != (PSS *)NULL)
{
 frozen_pss = recovery_pss->pspid;
 recovery_pss->pextstat |= PEXT_FREEZE_RECOVERY_THREAD;
}
V_SPINLOCK(Resource->ha_spin);
if (TRACECMDLINE(RECOVER, 56) && (frozen_pss != 0))
{
 scerrlog("We have set the FREEZE bit in recovery thread %d. \n",
    frozen_pss);
 }
 return;
}
/*
** REC_FREEZE_THREAD
**
** Purpose:
**  This function will be called when the caller sees the
** PEXT_FREEZE_RECOVERY_THREAD bit in pss. Then this thread will go to sleep
** on the assertion that the server is running with too many threads
** than it can handle.
**
**  Before calling this function, pss->pextstat was checked and
** PEXT_FREEZE_RECOVERY_THREAD was found set. Since this status was not
** checked under spinlock, check it again.
**
** Parameter:
** None.
**
** Return:
** None.
**
** Synchronization:
** The global status REC_INFO_TOO_MANY_THREADS is checked
** under spinlock.
**
** History:
** 4/02 (fzhou) - created
*/
void
rec_freeze_thread()
{
 LOCALPSS(pss);
 RECOVERY_INFO *recovery_info;
 recovery_info = &Resource->rrecovery_info;
 P_SPINLOCK(Resource->ha_spin);
 if (!(FREEZE_RECOVERY_THREAD(pss)))
 {
  SYB_ASSERT(!(recovery_info->status &
    REC_INFO_TOO_MANY_THREADS));
  V_SPINLOCK(Resource->ha_spin);
  return;
 }
 while (recovery_info->status &
    REC_INFO_TOO_MANY_THREADS)
 {
  V_SPINLOCK(Resource->ha_spin);
  /*
  ** Since this is not in a performance sensitive code path,
  ** no need for a mda identifier. Use UNUSED instead.
 */
 upsleepgeneric(SYB_EVENT_NON_STRUCT(&recovery_info->status),
                         (char *)&recovery_info->status,
                         sizeof(recovery_info->status),
                         REC_INFO_TOO_MANY_THREADS,
   FALSE, UNUSED);
 P_SPINLOCK(Resource->ha_spin);
 }
 /*
 ** Clear the PEXT_FREEZE_RECOVERY_THREAD bit in pss because there
 ** is no need to freeze any thread.
 */
 pss->pextstat &= ~(PEXT_FREEZE_RECOVERY_THREAD);
 V_SPINLOCK(Resource->ha_spin);
 return;
}
/*
** DORECOVER()
**
** Dorecover calls recovery on all databases in Sysdatabases
**      except for master, model, tempdb, sybsecurity, sybsystemdb and
**      sybsystemprocs. Note that user created temporary databases
** are also handled here.
**
** Parameters:
**    model_recovered -- TRUE if the model  database has been recovered
**    modeldbt     -- dbtable for the model database
**
** Returns:
**    none
**
** Side Effects:
**    -- all user databases recovered. User created temporary databases
**       are recovered only if the model database has been recovered.
**
** History:
**  Written 9/85 (cfr)
**  3/86  (jld) - changed outer loop to reopen sysdatabases
**   each time and search for next dbid to free up
**   a system sdes for use by undo.
**  8/86  (jkr) - open databases by id and get name for error
**   messages from dbtable.
**             6/88  (genew) - added ex_raise call if opendb() fails
*/
void
dorecover(SYB_BOOLEAN model_recovered, DBTABLE *modeldbt)
{
 int read_result;
 REC_ORDER_INFO *rec_order_info;
 REC_CALLER_ARG rec_caller_arg;
```

```
VOLATILE REC_CALLER_BKOUT_CTX backout_ctx;
LOCALPSS(pss);
/* keep backout variables in memory */
SYB_NOOPT(backout_ctx);
/* Initialize */
rec_order_info = NULL;
MEMZERO(&backout_ctx, sizeof(backout_ctx));
MEMZERO(&rec_caller_arg, sizeof(rec_caller_arg));
/* mark Pss so we can tell who is recovery */
pss->pstat |= P_ISRECOVERY;
DEBUG_SET(pss);
INSTALL_BACKOUT_HANDLES(&pss->pbkout_rec_caller, &backout_ctx,
    rec_caller_backout);
/*
** Set the global_cleanup indicator to TRUE because if any
** error happens, we want to clean up the resource we acquires
** at the end of rec__caller_hdlr().
** Model is locked by the caller and passed in.  Store it in the
** context structure
*/
backout_ctx.global_cleanup = TRUE;
if (model_recovered)
{
 backout_ctx.model_locked = TRUE;
 backout_ctx.modeldbt  = modeldbt;
}
if (ex_handle(EX_ANY, EX_ANY, EX_ANY, rec__caller_hdlr))
{
 return;
}
/*
** Fill the fields in the argument structure which will be used
** by lower level functions to recover user created temporary
** databases.
*/
rec_caller_arg.model_recovered = model_recovered;
rec_caller_arg.modeldbt = modeldbt;
/*
** Before we start the recovery of user databases, open master
** database so that master db will be the current database.
** Without doing this, the last db recovered will be the current
** database, which might be tempdb.
** We need to this because later when we are trying to do some
** initial prepare work for recovery, such as tuning the buffer
** pools in the default data cache, internal sql will be called,
** and it will start a transaction. If the last db recovered is
** tempdb, trying to start a multi-db transaction in tempdb will
** fail.
*/
if (!usedb((BYTE *) NULL, MASTERDBID, pss->psuid))
```

```
 ex_raise(OPENDBM, REC_RETURN, EX_CONTROL, 34);
backout_ctx.master_dbt = pss->pdbtable;
/*
** Get all dbs into rec_order_info by order that is specified and / or
** by dbid for dbs that don't have special recovery order.
*/
if ((read_result =
 rec_build_recovery_info((REC_CALLER_BKOUT_CTX *)&backout_ctx,
     Resource->rmasterdbt)) != -1)
{
 rec_order_info = Resource->rrecovery_info.rec_order_info;
}
/*
** If we have some databases to recover, but we failed to populate
** the rec_order_info structure, we will fall back to recover
** in serial following dbid order.
**
** Sysdatabases MASTER will be opened and closed in
** rec_getnextdb_by_dbid().
*/
if ((!rec_order_info) && (read_result != 0))
{
 rec__boot_recover_dbs((REC_CALLER_BKOUT_CTX *)&backout_ctx,
   &rec_caller_arg, READ_FAILED);
}
else if (Resource->rrecovery_info.status & REC_INFO_PARALLEL)
{
 /*
 ** Otherwise, as long as we have successfully populated the
 ** rec_order_info, and we have more than 1 db to recover (only
 ** then the status REC_INFO_PARALLEL will be set), call
 ** rec_run_parallel_recovery() to recover according to the
 ** rec_order_info in serial or parallel mode depending on
 ** the degree of parallelism for recovery.
 */
 rec_order_info->metadbid = MASTERDBID;
 rec_run_parallel_recovery((REC_CALLER_BKOUT_CTX *)&backout_ctx,
     &rec_caller_arg);
}
# ifdef TRACE_REC
if (TRACE(RECOVER, 20))
{
 SETTRACE(PAGEM, 20);
}
# endif
/* unmark pss as recovery's */
pss->pstat &= ~P_ISRECOVERY;
/*
** If there are no offline db's clean up syscoordinations of any rows
** inserted by recovery. Note that during upgrade syscoordinations
```

```
 ** may not exist yet. We are accounting for this case below. We
 ** are doing this cleanup only if sybsystemdb is recovered to
 ** avoid printing spurious messages to errorlog which may cause
 ** tech support calls.
 */
 if (SYSCOORDINATIONS_EXISTS &&
     database_recovered((BYTE *) "sybsystemdb", 11, 0) &&
     !(rec_exists_offlinedb()))
 {
  rec_delete_syscoord();
 }
 /* announce completion of recovery */
 ex_callprint(EX_NUMBER(RECOVER, REC_FINISHED), EX_INFO, 1);
 /*
 ** We are done with boot time recovery for user databases,
 ** call clean up function before we return.
 */
 CALL_CLEANUP_FUNC(&pss->pbkout_rec_caller);
 /* Upgrade from UP to MP */
 if (TRACECMDLINE(RECOVER,7))
 {
  if (!TRACECMDLINE(RECOVER, 8))
  {
   /* finally, upgrade the major number in the config
   ** block
   */
   Resource->rconfig->cmajor = 2;
   Resource->rconfig->cchecksum =
    cchksum((char *) Resource->rconfig,
            sizeof (DS_CONFIG) - sizeof (Resource->rconfig->cchecksum));
   confwrite(Resource->rconfig);
  }
  /* done with upgrade so shut down the server */
  if (!TRACECMDLINE(RECOVER, 8))  /* dry run */
   ucierrlog(NOFAC_SERVER, UTILS_UPGFINCHKERRLOG);
  else
   ucierrlog(NOFAC_SERVER, UTILS_TRIALUPGFINCHKERRLOG);
  ueshutdown(0);
 }
 return;
}
/*
** REC__BOOT_RECOVER_DBS
**
** This is the interface function to recover all user databases for boot
** time recovery for all different recovery modes. Depending on the passed
** in recovery mode, it will recover the databases by dbid order or according
** to the rec_order_info structure in Resource which was populated by the
** caller.
**
```

```
** Within a loop, it will call boot__recover_and_online() to recover all
** databases until there is no more db to recover, or it is instructed to
** stop.
**
** Parameter:
** backout_ctx - the backout structure which will have some
**   fields filled here in this function
** rec_caller_arg - the structure which contains the information
**   needed to recover user created tempdb.
** rec_mode - the choices for the rec_mode are:
**   READ_FAILED: Failed to populate the rec_order_info
**    structure, and therefore databases will
**    be recovered serially by dbid order.
**   SERIAL: Databases will be recovered serially according
**    to the populated rec_order_info structure.
**   PARALLEL: databases will be recovered concurrently
**    according to the populated rec_order_info.
**
** Return:
** Nothing.
**
** History:
** 1/2003 (fzhou) - created.
*/
SYB_STATIC void
rec__boot_recover_dbs(REC_CALLER_BKOUT_CTX *backout_ctx,
   REC_CALLER_ARG *rec_caller_arg, int rec_mode)
{
 dbid_t  curdbid;
 int  getdb_result;
 int  rec_order_num;
 SYB_BOOLEAN is_parallel_recovery;
 REC_ORDER_INFO *rec_order_info;
 RECOVERY_INFO *recovery_info;
 LOCALPSS(pss);
 /* Initialize local variables. */
 is_parallel_recovery = FALSE;
 rec_order_num = UNUSED;
 /*
 ** Set the global_cleanup indicator to FALSE because if any
 ** error happens in the scope of this function, we do not want to
 ** clean up the resource. Instead, the backout code will continue
 ** to the next database.
 */
 backout_ctx->global_cleanup = FALSE;
 recovery_info = &Resource->rrecovery_info;
 rec_order_info = recovery_info->rec_order_info;
 /*
 ** Initialize the start point of the scan to get next database.
 ** We need to do it here before the exception backout code, because
** after dealing with exceptions, we want the scan to start from
** the last database that was recovered (the scan start point will
** be set in the backout code).
**
** If we are recovering by dbid order (rec_mode == READ_FAIL),
** curdbid serves as the start point of scan of sysdatabases.
** Initialize it to MODELDBID so that the scan will start after
** model db.
*/
curdbid = MODELDBID;
/*
** Init backout vars that will be filled in this scope.
**
** we hopefully assert that backout_ctx->curdbid will be filled with a
** meaningful value before we can hit backout code.
**
** RESOLVE:  can we catch an exception (say, B_NOIO) due to access to
** master..sysdatabases?  If so, we'll mark some innocent database
** as suspect, because the current presumption is that if we caught
** an error, the problem was in the user database we're trying to
** recover.  We may need a "recovering_database" flag to examine
** in the backout code, which would be turned on when we open the
** database and turned off when we close it.  In that case, what
** would we do when we catch an error with the flag off?  Shut down
** recovery?  For now, we initialize backout_ctx->curdbid to zero,
** so that if this unlikely scenario plays out in the first pass
** through the loop, we'll fail in rec_dbsuspect() on a dbid of zero,
** rather than mark the model database as suspect.  However, such an
** error would more likely occur here when our loop accesses
** Sysdatabases where it has grown to a new segment, since Sysdatabases
** entries for the system databases have already been accessed
** successfully.
*/
backout_ctx->backout_dbt = (DBTABLE *)NULL;
backout_ctx->curdbid = 0;
backout_ctx->rec_order_num = UNUSED;
if (ex_handle(EX_ANY, EX_ANY, EX_ANY, rec__caller_hdlr))
{
 /*
 ** The current database is not recoverable.
 ** Make sure the suspect bit is set.
 **
 ** If dbt NULL due to failed open, he can cope.
 ** Need this here because error could be raised
 ** before recovery() is called, for example,
 ** when we try to open the dbtable.
 */
 rec_dbsuspect(backout_ctx->curdbid, backout_ctx->backout_dbt);
 if (backout_ctx->backout_dbt)
 {
  backout_ctx->backout_dbt = (DBTABLE *)NULL;
  closedb(USEPREV);
 }
 CLOSE_SDES(&backout_ctx->db_sdes);
 /*
 ** If we haven't cleaned up the rec_order_item yet,
 ** clean it up.
 */
 if (backout_ctx->rec_order_num)
 {
  rec_order_num = backout_ctx->rec_order_num;
  backout_ctx->rec_order_num = 0;
  rec_cleanup_recovery_item(rec_order_num, FALSE);
 }
 /*
 ** simply return and stop recovery if there is any error
 ** happened before we recover the first database.
 */
 if (!backout_ctx->curdbid)
 {
  /*
  ** Set the global_cleanup indicator to TRUE before we
  ** return to the caller, because if any error happens
  ** in the caller, we want to clean up the resource.
  */
  backout_ctx->global_cleanup = TRUE;
  return;
 }
 else
 {
  /* print out the 'continue' message */
  ex_callprint(EX_NUMBER(RECOVER2, REC_CONTINUE_NEXTDB),
    EX_INFO, 3);
  /*
  ** If we are recovering by dbid, always reset the
  ** start point of the scan of sysdatabases to the
  ** dbid of the last db recovered so that recovery
  ** could fall through to the next db.
  */
  curdbid = backout_ctx->curdbid;
 }
}
if (rec_mode == READ_FAILED)
{
 /*
 ** Since we failed to build the rec_order_info structure
 ** in memory, the rec_order_num will not be used. Although
 ** the backout_ctx structure has been MEMZERO'd earlier,
 ** explicitly set the rec_order_num field to UNUSED.
 */
 backout_ctx->rec_order_num = UNUSED;
 while (getdb_result = rec_getnextdb_by_dbid(backout_ctx,
    &curdbid, MASTERDBID) !=
      REC_GOT_NOMORE_DB)
 {
  backout_ctx->curdbid = curdbid;
  /*
  ** If this database is a user created temporary
  ** database, call the special function to recover,
  ** then continue to the next database.
  */
  if (getdb_result == REC_GOT_TEMPDB)
  {
   /*
   ** Set the backout_dbt in the backout context
   ** structure to NULL just in case there is an
   ** exception while recovering a temporary
   ** database. We never have the DBTABLE for a
   ** temporary database since all relevant
   ** happens under recover_tempdb().
   */
   rec__user_tempdb(rec_caller_arg);
   continue;
  }
  boot__recover_and_online(backout_ctx);
  /*
  ** Before we get the next database, yield if we have no
  ** time.
  */
  TIMESLICE_YIELD(pss);
 }
 goto exit_point;
}
SYB_ASSERT((rec_mode == PARALLEL) || (rec_mode == SERIAL));
if (rec_mode == PARALLEL)
{
 is_parallel_recovery = TRUE;
 P_SPINLOCK(Resource->ha_spin);
}
/*
** Keep getting the next database to recover until we are told not to
*/
while (TRUE)
{
 getdb_result = rec_getnextdb_to_recover(backout_ctx);
 if ((getdb_result == REC_NEED_TO_EXIT) ||
     (getdb_result == REC_WAKEUP_FROZEN_THREAD) ||
     (getdb_result == REC_GOT_NOMORE_DB))
 {
  break;
 }
 if (is_parallel_recovery)
 {
  V_SPINLOCK(Resource->ha_spin);
 }
 SYB_ASSERT ((getdb_result == REC_GOT_NEXT_DB) ||
      (getdb_result == REC_GOT_TEMPDB) ||
      (getdb_result == REC_ONLINE_DB) );
 if (getdb_result == REC_GOT_TEMPDB)
 {
  rec__user_tempdb(rec_caller_arg);
 }
 else if (getdb_result == REC_ONLINE_DB)
 {
  /*
  ** Make the current db accessible (i.e. online).
  */
  rec_makedb_accessible();
 }
 else
 {
  /*
  ** For not recovered normal user database, go ahead and
  ** recover using normal path.
  */
  SYB_ASSERT(getdb_result == REC_GOT_NEXT_DB);
  boot__recover_and_online(backout_ctx);
 }
 /*
 ** Clean up the recovery item. Passing TRUE as we
 ** are in normal clean up as opposed to exception handling
 ** clean up.
 */
 rec_order_num = backout_ctx->rec_order_num;
 backout_ctx->rec_order_num = 0;
 rec_cleanup_recovery_item(rec_order_num, TRUE);
 /*
 ** Before we get the next database, yield if we have no
 ** time.
 */
 TIMESLICE_YIELD(pss);
 /*
 ** If this is invoked by spawned recovery threads, obtain the
 ** ha_spin before searching through the global structure again.
 */
 if (is_parallel_recovery)
 {
  P_SPINLOCK(Resource->ha_spin);
 }
}
```

```
 /*
 ** If rec_getnextdb_to_recover() found that server was handling too
 ** many threads than it could handle, and therefore decided to
 ** exit to reduce the number of running recovery threads:
 **
 ** 1. Clear the status indicating server was handling too many threads
 ** because the number of running recovery threads will be decremented
 ** right after this thread returns;
 ** 2. wake up any thread that could have been put to sleep.
 */
 if (getdb_result == REC_WAKEUP_FROZEN_THREAD)
 {
  SYB_ASSERT(Resource->rrecovery_info.status & REC_INFO_PARALLEL);
  /*
  ** clear this status because this thread is going to
  ** exit and therefore reduce the number of running threads.
  */
  Resource->rrecovery_info.status &=
    ~(REC_INFO_TOO_MANY_THREADS);
  /* This should always be true. */
  if (is_parallel_recovery)
  {
   V_SPINLOCK(Resource->ha_spin);
  }
  (void) upwakeup(SYB_EVENT_NON_STRUCT(&Resource->rrecovery_info.status));
 }
 else
 {
  /* This is only a sanity check on the return value. */
  SYB_ASSERT((getdb_result == REC_NEED_TO_EXIT) ||
      (getdb_result == REC_GOT_NOMORE_DB));
  if (is_parallel_recovery)
  {
   V_SPINLOCK(Resource->ha_spin);
  }
 }
exit_point:
 backout_ctx->curdbid = 0;
 backout_ctx->rec_order_num = 0;
 /*
 ** Set the global_cleanup indicator to TRUE before we return to
 ** the caller, because if any error happens in the caller, we want to
 ** clean up the resource.
 */
 backout_ctx->global_cleanup = TRUE;
 return;
}
/*
** RECOVERY
**
```

```
** Description:
**
**  The recovery system is the part of SQL Server which masks faults.
** A good description of what this means can be found in _The Log Recovery
** Module_ (/calm/svr/docs/doc/arch/design/low_level/dbaccess/logrec.fm)
**
** Recovery uses the transaction log (SYSLOGS) to restore the database
** to a consistent state.
**
** This module runs recovery over a portion of the database's log.
** It is the entry point to the recovery system. It is passed a token
** indicating the circumstances in which it is called, one of:
**   REC_INIT When SQL Server starts up
**   REC_LOADDB After the database has been restored as part of a 'load
**   database' command.
**   REC_LDXACT After part of the log has been restored as part of a
**   'load tran' command.
**
** There are four stages of recovery
** 1) Analysis pass
**  This reads through all of the current log.
**  The transaction table is constructed.
** 2) Redo Pass
**  All log records in the current log are redone.
** 3) Undo nest Top Actions.
**  Any incomplete Nested Top Actions are undone.
**  The database is now structurally consistent.
** 4) Undo Pass
**  The log is read backwards, and records belonging to
**  incomplete transactions are undone.
**
** Depending on the type of recovery appropriate routines will be called.
**
**
** NOMENCLATURE NOTE:  variables and comments within this code contain
** the string "quiesce", as in "QUIESCE DATABASE" and "quiescentpt_info".
** Two different concepts are represented by these words, which are only
** somewhat related:
**
** 1) QUIESCE DATABASE is a SQL command which freezes updates to
**  one or more databases to provide the user the chance to clone
**  the database device(s) via some external OS commnand.  During
**  REC_INIT, recovery must be cognisant of the special needs
**  of such a cloned database, when it may be required to emulate
**  the product of LOAD DATABASE.
**
** 2) A reference to a "quiescent point" discusses an internal
**  feature of LOAD TRANSACTION of a log dump produced by DUMP
**  TRANSACTION WITH STANDBY_ACCESS. This internal feature
**  establishes a place in the log where no transactions are
**  active, a point up to which recovery will proceed, requiring
**  no undo phase with its unwanted compensation log records.
**
** Further confusion comes from interplay between these two concepts,
** where a database cloned under the aegis of QUIESCE DATABASE may
** require a recovery treatment similar to that for a standby access
** dump tran, in that the undo phase would be avoided and the database
** automatically brought online for standby access.
**
**
** Parameters
** input_rectype   - type of recovery, one of:
**
**     REC_INIT
**     REC_LDXACT
**     REC_LOADDB
**
** num_openxacts_tofill - number of open (incomplete) transaction
**      that were found by the analysis pass. This
**      routine populates this variable for use
**      by the caller. This could be a NULL pointer.
** Returns
** TRUE -- success
** FALSE -- failure
**
** Assumptions:
** Recovery assumes that the database is open and current.
**
** MP Synchronization
** dbt_stat in the dbtable for the current dbtable is read WITHOUT
** the dbt manager spinlock, requiring that dbt_stat be an ATOMINT.
** The dbtable manager is called to turn off the DBT_NOTREC bit
** under its spinlock, once the database has been recovered.
**
** History
** written 08/17/85 (cfr)
** major changes 09/10/85 (cfr)
** revised 08/20/87 (ht) - modified algorithm to recover
**  transactions in the order of their ENDXACT records rather then
**  the order of their completion numbers.
** 5/10/89 (jkr) mp sync changes
** 7/21/93 (laxmi)  Pass a special status of DBCKPT_REBOOT to
** checkpoint(), when recovery is initiated by a reboot and it is
** permissible to  write a checkpoint record.  This will set
** CKPT_REBOOT flag in the checkpoint record.
** 1/10/96 (psalding) Re-read the systhresholds table in order
**    to refresh the in-memory thresholds cache.
** Autumn 1996 (asherman) Rewritten for CLR based recovery.
** Winter 1996 (kannan)    Split this into recovery() and rec_onlinetime()
**    for clarity sake.
```

```
*/
int
recovery(int input_rectype, int *num_openxacts_tofill)
{
 int  local_rectype;
 int  quiescentpt_info;
 DBTABLE  *dbt;  /* dbtable for db being recovered */
 SDES  *logsdes; /* for dbinfo operations */
 XTABLE  *xtable; /* transaction table holder */
 REC_FP_TABLE *fptab;
 RECTABLE *rectable; /* recovery resource table */
 VBITMAP  *allocbitmap; /* bitmap to keep track of allocs */
 XDES  *xdes;  /* to access log  */
 DBINFO  *dbiptr; /* points to dbinfo  */
 ITAG  itagdbinfo; /* to access dbinfo  */
 ITAG  *itagp;  /* to facilitate exception handling */
 dbid_t  dbid;  /* dbid of db being recovered */
 int16  ldstate; /* dbinfo status bits */
 XCKPT  ckpt;  /* Current Checkpoint log record */
 XLRMARKER ckptlr;  /* Marker of Checkpoint log record */
 XLRMARKER true_ckptlr; /* The true checkpoint marker. */
 XLRMARKER firstlogmrkr; /* First log rec in scans  */
 XLRMARKER lastlogmrkr; /* Last log rec in scans   */
 XLRMARKER true_lastlogmrkr;
    /* True last log record marker. */
 REC_ANALYSIS analysis; /* filled by analysis pass  */
 int            undo_recs;      /* # of records undo pass will read */
 int32  dop_status; /* Whatever we want the three recovery
    ** passes to add to their DOPARAMS.
    */
 SYB_BOOLEAN     masterupgradedone; /* Master has been upgraded  */
 SYB_BOOLEAN     skip_undopass;  /* We should skip the undo pass.*/
 SYB_BOOLEAN     in_loadseq;     /* We are in a load sequence.   */
 SYB_BOOLEAN     pre_clr_log;    /* We are dealing with log from
    ** release prior to the introduction of
    ** compensation log records in server.
    */
 SYB_BOOLEAN     trace_simulate_fptable_oflow;
 BASIC_MESSAGE_PARAMS    quiescedb_msg_params;
 LOCALPSS(pss);
 VOLATILE REC_BKOUT_CTX copy;
 /* keep backout variables in memory */
 SYB_NOOPT(copy);
 /* initialize variables that may have to be backed out */
 MEMZERO(&copy, sizeof (copy));
 copy.check_freespace = TRUE;
 copy.xdes_endstat = TRUE;
 copy.pss = pss;
 /* Initialize the error return status. */
 pss->pretstat = 0;
```

```
/* simplify db references */
dbid = pss->pcurdb;
dbt = pss->pdbtable;
/* Initialize local variables */
local_rectype = input_rectype;
itagp = &itagdbinfo;
rectable = (RECTABLE *)NULL;
 masterupgradedone = FALSE;
skip_undopass = FALSE;
in_loadseq = FALSE;
pre_clr_log =
 ((dbt->dbt_logvers < UNDO_LOGS_CLRS_LOGVERS_ID) ? TRUE : FALSE);
trace_simulate_fptable_oflow = TRACECMDLINE(RECOVER, 48) ? TRUE : FALSE;
MEMZERO(&quiescedb_msg_params, sizeof(quiescedb_msg_params));
dop_status = 0;
# ifdef TRACE_REC
if (TRACECMDLINE(RECOVER, 0))
{
 TRACEPRINT("RECOVERY: dbid %d\n", dbid);
}
# endif
/*
** Install the backout context structure and the backout cleanup
** function on pss.
*/
INSTALL_BACKOUT_HANDLES(&pss->pbkout_rec, &copy, rec__backout);
if (ex_handle(EX_ANY, EX_ANY, EX_ANY, rec_handle))
{
 rec__ctx_cleanup((REC_BKOUT_CTX *)&copy);
 /* If cant recover master, exit, otherwise mark db as suspect */
 if (copy.pss->pcurdb == MASTERDBID)
 {
  ex_callprint(EX_NUMBER(RECOVER, REC_BADMASTER), EX_INFO, 1);
  ueshutdown(0);
 }
 else
 {
  rec_dbsuspect(copy.pss->pcurdb, copy.pss->pdbtable);
 }
 /* Mark that we are no longer in recovery */
 copy.pss->pstat &= ~P_ISRECOVERY;
 /*
 ** Before returning, clear the backout data in pss.
 */
 CLEAR_BACKOUT_HANDLES(&copy.pss->pbkout_rec);
 return (FALSE);
}
        /*
** Open an SDES to access the dbinfo. SYSLOGS is the only table
** we can open at the moment, yet it happens to be just the
** very sdes required to get the dbinfo.
**
** This SDES is not used for access to SYSLOGS.
** Access to SYSLOGS should only be through the XLS.
      */
logsdes = OPEN_SYSTEM_TABLE(SYSLOGS, (dbid_t) UNUSED, dbt);
copy.logsdes = logsdes;
if (local_rectype == REC_INIT)
{
 /*
 ** For crash recovery, disable lock caching in the recovery pss.
 ** During crash recovery, the recovery task acquires locks on
 ** behalf of prepared external transactions. These locks should
 ** not be cached in the recovery pss because these locks belong
 ** to the transaction and not the recovery task.
 **
 ** Note that if rectype is converted to REC_LOADDB by our
 ** recovering an in-quiesce database, then there will be no
 ** transactions in prepare state, thanks to enforcement by
 ** QUIESCE DATABASE HOLD.
 **
 ** RESOLVE:  okay, then, should we move this block to after the
 **          rec_quiescedb_state() call?
 */
 SET_LOCK_CACHING_OFF(pss);
}
/* Going forward, should recovery type be different? */
if (local_rectype == REC_INIT)
{
 int crash_in_ldtran;
 /*
 ** Open an XDES to be used for log access.
 */
       xdes = xact_begin_session(dbt);
 (void) xls_open(xdes, pss, XLS_READONLY, 0);
 /*
 ** Check for crash in load xact.  CRASHED_IN_LOAD_RECOVERY means
 ** we crashed in the recovery phase of load xact; in that
 ** case, we must recover as LOAD TRANSACTION. Having crashed
 ** in the load phase is repaired by the routine we're calling,
 ** all we need to do is to get the latest log marker. In
 ** CRASHED_IN_LOAD case, the user should be able to re-load
 ** the failed dump tran.
 */
             crash_in_ldtran = ldx_checkldstate(xdes);
 if (crash_in_ldtran == CRASHED_IN_LOAD)
 {
  xls_getmarker(xdes, XLSM_LOGLAST, TRUE, &lastlogmrkr);
 }
 else if (crash_in_ldtran == CRASHED_IN_LOAD_RECOVERY)
 {
  local_rectype = REC_LDXACT;
 }
 (void) xls_close(xdes, FALSE);
 xact_end_session(xdes);
}
# ifdef TRACE_REC
/* test exception handling by raising one */
if (TRACE(RECOVER, 4))
{
 if (dbid > MODELDBID)
 {
  TRACEPRINT(
   "RECOVERY: raise fake dbsuspect exception\n");
  ex_raise(RECOVER, REC_RETURN, EX_CONTROL, 18);
 }
}
/* test exception handling for master */
if (TRACE(RECOVER, 5))
{
 if (dbid == MASTERDBID)
 {
  TRACEPRINT(
  "RECOVERY: raise fake dbsuspect exception on master\n");
  ex_raise(RECOVER, REC_RETURN, EX_CONTROL, 19);
 }
}
# endif
 /*
 ** Initialize the suspect information, if any.
** The dbt->ha_suspect_info pointer gets allocated in this
** procedure, if the suspect granularity is other than
** database. Note that for load transaction, this step
** is skipped because this will have already happened
** during boot or loadtime recovery.
 */
 if ((local_rectype != REC_LDXACT) && (!sg_readdisk_suspectinfo(dbt)))
  ex_raise(RECOVER, UT_RETURN, EX_CONTROL, 30);
/* Open an XDES to be used for log access within this function. */
copy.xdes = xdes = xact_begin_session(dbt);
(void) xls_open(xdes, pss, XLS_READONLY, 0);
/* Perform any required embedded pre-recovery upgrade functions. */
rec_init_upgrade(xdes);
/* Initialize the transaction table */
if (!(xtable = create_xtable()))
 ex_raise(RECOVER, XACTOFLOW, EX_DBFATAL, 5, 0, 0);
copy.xtable = xtable;
/*
** If this is crash recovery create the recovery resource table
** The resources acquired by push transactions will be kept track
** of by the recovery task in this table.  This is used for releasing
** resources acquired when an error is encountered and also to keep
** track of the address locks on resinstantiated transactions
** which have to be held  till the database is onlined
*/
if (local_rectype == REC_INIT)
{
 if (!(rectable = create_rectable()))
 {
  ex_raise(RECOVER, XACTOFLOW, EX_DBFATAL, 2, 0, 0);
 }
 pss->prectable = rectable;
 copy.rectable = rectable;
}
/*
** Create flushed pages table only when:
**
** PFTS during recovery time is not disabled
**  AND
** the caller is boot-time recovery
**
** Note that REC_INIT can subsequently be converted to REC_LOADDB
** if we're recovering for QUIESCE DATABASE, but then, unlike
** recovering for an actual LOAD DATABASE, PFTS records reflect
** reliable flushes to the database devices, so "fptab" will
** still be usable and valuable.
*/
fptab = NULL;
if (RECTIME_PFTS_ENABLED(dbt) && (local_rectype == REC_INIT))
{
 /* Try to instantiate table unless trace flag says not to. */
 if (!trace_simulate_fptable_oflow)
 {
  /* Could be NULL if out-of-memory. */
  fptab = create_recfptable();
 }
 copy.fptab = fptab;
}
if (!fptab)
{
 /* Downstream recovery fns have no fptable to load or consult */
 dbt->pfts_data.pfts_status |= RECTIME_NO_FPTAB;
}
/*
** Create a variable bitmap.
**
** Each bit in the map corresponds to one allocation page. This map
** is used during recovery to keep track of the allocation pages
** requiring a cleanup of its deallocation bits in its extents.
** The bit for an allocation page is SET if we see a dealloc type
** record corresponding to that allocation page.
*/
allocbitmap = vbit_mapcreate(dmap_high_lpage(DBT_DISKMAP(dbt)));
       copy.allocbitmap = allocbitmap;
/*
** Find the markers for recovery passes using the following rules
** to decide how to pick up the first and last log markers.
**
** a) If the dump that was loaded is a transaction dump made
**    for standby mode (aka pseudo replication) then we need
**    to calculate the quiescent log marker for the starting
**    and ending point of recovery.
**
** b) If the dump being loaded (recovered) is a normal dump
**    then no need to calculate quiescent point.
*/
ITAGINIT(itagp);
dbiptr = ind_dbinfoget(logsdes, itagp);
ldstate = dbiptr->dbi_ldstate;
ind_dbinforelease(itagp);
if (ldstate & DBI_PREVLOAD_STANDBYXACTDMP)
{
 quiescentpt_info = QUIESCENT_FIRSTLOG_MARKER;
 /*
 ** If we are doing standby mode load tran recovery,
 **  OR
 ** if we are recovery a database that was online'ed for
 ** standby mode access and we are not in load-tran recovery
 ** of a normal dump, then we need to calculate the quiescent
 ** last log marker, as we don't want to recover all the way
 ** till the actual end of the log, as that might add log
 ** records and jeopardize the load sequence.
 */
 if (ldstate & DBI_CURLOAD_STANDBYXACTDMP)
 {
  quiescentpt_info |= QUIESCENT_LASTLOG_MARKER;
 }
 else
 {
  if ((dbt->dbt_stat2 & DBT2_ONL_STANDBY) &&
      (local_rectype != REC_LDXACT))
  {
   quiescentpt_info |= QUIESCENT_LASTLOG_MARKER;
  }
 }
}
else
{
 quiescentpt_info = NO_QUIESCENT_MARKER;
}
```

```
/*
** Instantiate the boundary log markers.
** If the trace flag is set, tell the humans what the values are.
*/
rec_logbounds(local_rectype, dbt, logsdes, &ckpt, &ckptlr,
       &firstlogmrkr, &lastlogmrkr, quiescentpt_info);
if (TRACECMDLINE(RECOVER, 25))
{
              TRACEPRINT("recovery: Database (%d), recovery type (%d)\n",
                                   dbt->dbt_dbid, local_rectype);
 xlm_printmarker("recovery:firstlogmrkr=%s\n", &firstlogmrkr);
 xlm_printmarker("recovery:lastlogmrkr=%s\n", &lastlogmrkr);
 xlm_printmarker("recovery:ckptlr=%s\n", &ckptlr);
}
/*
** If DB is in quiesce state, this function might patch the db to look
** like LOAD DATABASE just happened. In that case, rectype will change
** from REC_INIT to REC_LOADDB.
**
** Later code says that lastlogmarker can be changed, but only for
** certain obsolete log versions and only during LOAD TRAN recovery.
** Those obsolete log versions happen to precede the first ASE version
** to implement QUIESCE DATABASE, and anyway we need REC_INIT to enter
** this if-block.
*/
if (local_rectype == REC_INIT)
{
 local_rectype = rec_quiescedb_state(logsdes, &ckptlr,
        &lastlogmrkr,
        &quiescedb_msg_params);
 if (local_rectype == REC_LOADDB)
 {
  /*
  ** At least one per-logop function needs to know this.
  **
  ** A purer approach would have been to create a new
  ** rectype, but it was seen as to risky to implement
  ** late in a development cycle, and messier for the
  ** the REC_INIT and REC_LOADDB testers, all but one of
  ** whom didn't care about the intermediate rectype
  ** created by rec_quiescedb_state() but all of which
  ** would then have to test for two types.
  **
  ** The immediate customer for the hybrid "is-REC_LOADDB-
  ** but-was-REC_INIT" recovery type in a PFTS assertion,
  ** which needs assurance that the flush represented by
  ** a PFTS record truly reached the currently operative
  ** db devices, a condition not guaranteed in recovery
  ** of an actual load database.
  */
  dop_status |= DOP_LOADDB_FROM_QDB_INIT;
 }
}
/* Initialize analysis structure */
MEMZERO(&analysis, sizeof(analysis));
/*
** Save the true last log and checkpoint markers for pre-CLR
** LOAD TRAN recovery. This is necessary because it is possible
** that the analysis, redo and undo passes will be run against
** sections of the log (subsets of the log records between
** 'firstlogmrkr' and 'lastlogmrkr') rather than running the passes
** on all these log records at one time.
*/
if ((pre_clr_log) && (local_rectype == REC_LDXACT))
{
 xlm_assignmarker(&lastlogmrkr, &true_lastlogmrkr);
 xlm_assignmarker(&ckptlr, &true_ckptlr);
}
if (TRACECMDLINE(RECOVER, 47))
{
 /* For PFTS, measure recovery performance numbers. */
 scerrlog("recovery(): PFTS ANALYSIS Started.\n");
}
/* Now we need a row buffer to pass to other callers */
copy.rowbuf = GET_ROW_BUFFER(pss);
next_log_section:
/*
** ANALYSIS PASS.
**
** Rec_analyze_log() may modify the value of 'lastlogmrkr' in two
** cases:
**
** 1)   For a very old log version, analysis pass has determined that
**      the scope of recovery should be reduced to avoid an incompatible
**      sort algorithm.
**
** 2)   For pre-CLR logs during LOAD TRAN recovery, a CHECKPOINT log
**      record has been found which was written during a server
**      re-start. This is the only chance of looping to
**      "next_log_section".
*/
rec_analyze_log(local_rectype, dbt, xtable, fptab, &firstlogmrkr,
  &lastlogmrkr, &ckptlr, &analysis, allocbitmap,
  (BYTE *) copy.rowbuf, dop_status);
/* Fill-in the count of open transactions, for use by the caller */
if (num_openxacts_tofill)
 *num_openxacts_tofill = xtable->items;
/*
** Check to see if this pss has been frozen. If so, call
** rec_freeze_thread to go to sleep.
```

```
 ** Do not use spinlock protection to check the status in pss
 ** for performance reason. The status will be checked again
 ** inside of rec_freeze_thread() under spinlock.
 */
 if (FREEZE_RECOVERY_THREAD(pss))
  rec_freeze_thread();
#if SANITY
 /*
 ** Debugging Aid - Print the analysis pass info.
 **     Print transaction table contents.
 */
 if (TRACECMDLINE(RECOVER, 25))
 {
  TRACEPRINT("dbid=%d, tot.logrecs=%d, tot.openxacts=%d\n",
   dbt->dbt_dbid, analysis.an_tot_recs, xtable->items);
  /*
  ** Print the information gathered during
  ** analysis pass, only when using diagserver.
  */
  print_analysis(&analysis, "recovery: ");
  /* print transaction table contents */
  print_xtable(xtable, "Recovery ");
 }
#endif
 /*
 ** PFTS_RESOLVE
 ** Debugging only
 */
 if (TRACECMDLINE(RECOVER, 49))
 {
  print_fptable (fptab);
 }
 /*
 ** PFTS Debugging Aid - To measure recovery performance numbers.
 */
 if (TRACECMDLINE(RECOVER, 47))
 {
  scerrlog("recovery(): PFTS ANALYSIS Finished/REDO Started.\n");
 }
 /*
 ** REDO PASS
 */
 undo_recs = rec_redo_log(local_rectype, dbt, logsdes, xtable,
    fptab, rectable, &firstlogmrkr, &lastlogmrkr,
    &ckptlr, allocbitmap, analysis.an_tot_recs,
    (BYTE *) copy.rowbuf, dop_status);
 /*
 ** The redo pass has marked all extents which have either
 ** been deallocated by completed transactions or which have
 ** pages deallocated by completed transactions. It is safe
** now to cleanup extents touched only by completed
** transactions.
*/
pg_zap_dealbit_db(allocbitmap, dbt,
   EX_DEALL_COMPLETED_XACT, local_rectype);
/*
** Check to see if this pss has been frozen. If so, call
** rec_freeze_thread to go to sleep.
** Do not use spinlock protection to check the status in pss
** for performance reason. The status will be checked again
** inside of rec_freeze_thread() under spinlock.
*/
if (FREEZE_RECOVERY_THREAD(pss))
 rec_freeze_thread();
/*
** Initialize the global time stamp for this database;
** after this point new log records may be written.
*/
if (local_rectype == REC_INIT)
{
 /*
 ** During boot time recovery if the checkpoint record is the
 ** last record in the log and has no active transactions i.e
 ** is the first log record that recovery scanned too, the highest
 ** timestamp is either the TS recorded in checkpoint or the timestamp
 ** on the last log page. Pass down this info to db_hights.
 ** Otherwise db_hights looks at every log record in the page
 ** and may not be able to translate some of them even after
 ** a clean shutdown.
 */
 if ((xlm_cmpmarker(&lastlogmrkr, &ckptlr) == 0) &&
   (xlm_cmpmarker(&firstlogmrkr, &ckptlr) == 0))
 {
  db_hights(dbt, &ckpt.xts);
 }
 else
 {
  db_hights(dbt, (DBTS *) NULL);
 }
}
else
{
 db_hights(dbt, (DBTS *) NULL);
}
/*
** PFTS Debugging Aid - To measure recovery performance numbers.
*/
if (TRACECMDLINE(RECOVER, 47))
{
 scerrlog("recovery(): PFTS REDO Finished.\n");
```

```
}
/*
** We are in a load sequence, if the last logrecord present
** in the log is same as the one maintained in the dbinfo.
**  AND
** the database wasn't made online before.  This test must
** happen only after the call to rec_quiescedb_state().
**
** The master database never goes offline.
*/
if ((lddb_test_inload(dbt)) &&
    ((!(dbt->dbt_stat2 & DBT2_AUTO_ONL)) &&
   (dbt->dbt_dbid != MASTERDBID)))
{
 in_loadseq = TRUE;
}
/*
** If we are doing boot-time recovery or recovering
** with a log from a release prior to the introduction
** of compensation log records, then we may have to run
** undo-pass also. Check that and do the needful.
*/
if ((local_rectype == REC_INIT) || (pre_clr_log))
{
 /*
 ** Skip the undo-pass, if the database is in the middle
 ** of a load sequence OR if the database was online'ed
 ** before for standby mode access.
 */
 if ((in_loadseq) || (dbt->dbt_stat2 & DBT2_ONL_STANDBY))
 {
  skip_undopass = TRUE; /* not safe to do undo pass */
 }
 /*
 ** Call the undo-pass, if necessary.
 **
 ** Note: if we don't have to worry about writing CLRs,
 **  we can afford to call the undo-pass even
 **  if we are in the middle of a load sequence.
 */
 if ((skip_undopass == FALSE) || (pre_clr_log))
 {
  /*
  ** Refresh the supergam array. Sysgams may have changed
  ** during redo, and it is important that the supergam
  ** array reflect the real page chain. We must do this
  ** before the undo pass(es) as CLRs will be logged.
  ** Note that if this call fails, then a message will
  ** have been printed into the errorlog by
  ** pg_fill_supergam() and the supergam array will have
** been marked invalid (so that cached page numbers
** will not be used).
*/
pg_invalidate_supergam(dbt);
pg_fill_supergam(dbt);
/*
** For CLR logs, undo incomplete top actions
** and refresh lastlogmrkr as CLRs may
** have been written. This is not required for
** pre-CLR logs, which have no concept of nested
** top actions.
*/
if (!pre_clr_log)
{
 rec_undo_incomplete_topactions(local_rectype,
     dbt, xtable);
 (void) xls_getmarker(xdes, XLSM_LOGLAST,
    TRUE, &lastlogmrkr);
}
/*
** UNDO PASS
*/
rec_undo_log(local_rectype, dbt, xtable, fptab,
                                   &firstlogmrkr, &lastlogmrkr, allocbitmap,
      undo_recs, (BYTE *) copy.rowbuf, dop_status);
/*
** Note that we should now cleanup extents which have
** deallocations due to open transactions as well as
** completed transactions. At this point recovery is
** done and the extents are in the state that they
** were at runtime.
*/
pg_zap_dealbit_db(allocbitmap, dbt,
    EX_DEALL_ALL, local_rectype);
/*
** Check to see if this pss has been frozen. If so,
** call rec_freeze_thread to go to sleep.
** Do not use spinlock protection to check the status
** in pss for performance reason. The status will be
** checked again inside of rec_freeze_thread() under
** spinlock.
*/
if (FREEZE_RECOVERY_THREAD(pss))
 rec_freeze_thread();
/*
** For LOAD TRAN recovery of pre-CLR logs, if the
** true (original) last log marker differs from
** the current last log marker....
*/
if ((pre_clr_log)
&&  (local_rectype == REC_LDXACT)
 &&  (xlm_cmpmarker(&lastlogmrkr, &true_lastlogmrkr)
   != 0))
{
 /*
 ** ....then the analysis pass found a
 ** CHECKPOINT log record which was written
 ** at reboot, so that it will have modified
 ** the lower boundary of recovery, in order
 ** to perform recovery only up to this
 ** CHECKPOINT log record. This is to emulate
 ** the events during run-time by rolling back
 ** any incomplete transactions at the time
 ** of the reboot, rather then waiting until
 ** the end of the log to roll these back.
 ** If there are remaining log records which
 ** still need to be recovered, set the upper
 ** boundary of recovery to the point that
 ** has been reached, and restore the lower
 ** boundary of recovery to the true (original)
 ** lower boundary, before repeating the
 ** analysis, redo and undo passes.
 */
 if (TRACECMDLINE(RECOVER, 25))
 {
  xlm_printmarker("RECOVERY: completed log section to %s\n", &lastlogmrkr);
 }
 /*
 ** Make the current last log marker, the
 ** point at which recovery will start for
 ** the next log section as well as the
 ** checkpoint marker, and re-establish the
 ** last log marker as the true (original)
 ** last log marker.
 */
 xlm_assignmarker(&lastlogmrkr, &firstlogmrkr);
 xlm_assignmarker(&lastlogmrkr, &ckptlr);
 xlm_assignmarker(&true_lastlogmrkr,
    &lastlogmrkr);
 /* Destroy and re-create the xact table. */
 copy.xtable = (XTABLE *) NULL;
 destroy_xtable(xtable);
 if (!(xtable = create_xtable()))
 {
  ex_raise(RECOVER, XACTOFLOW, EX_DBFATAL, 3, 0, 0);
 }
 copy.xtable = xtable;
 /* Destroy and re-create the vbit map. */
 copy.allocbitmap = (VBITMAP *) NULL;
 VBITMAPDESTROY(allocbitmap);
 allocbitmap = vbit_mapcreate(dmap_high_lpage(DBT_DISKMAP(dbt)));
        copy.allocbitmap = allocbitmap;
 /* Initialize analysis structure */
 MEMZERO(&analysis, sizeof(analysis));
 /*
 ** Repeat the analysis, redo and undo passes
 ** on the remainder of the log.
 */
 goto next_log_section;
}
/* checkpoint the database if it is safe */
if (!(dbt->dbt_stat & DBT_NOCKPT))
{
 /*
 ** Refresh the supergam array. Sysgams may have changed
 ** during recovery, so it is important that the supergam
 ** array reflect the real page chain. Note that if
 ** this call fails, then a message will have been printed
 ** into the errorlog by pg_fill_supergam() and the
 ** supergam array will have been marked invalid (so that
 ** cached page numbers will not be used).
 */
 pg_fill_supergam(dbt);
 /*
 ** Print message to errorlog to indicate the
 ** start of checkpoint.
 */
 mnt_ex_print(EX_NUMBER(RECOVER2, REC_CKPT_START), EX_INFO, 1,
   dbt->dbt_dbnlen,
   dbt->dbt_dbname);
 if (TRACECMDLINE(RECOVER, 57) &&
   (local_rectype == REC_INIT))
 {
  scerrlog("Before checkpointing '%.*s', the number of writes done by this thread %d is %d.\n",
    dbt->dbt_dbnlen,
    dbt->dbt_dbname,
    pss->pspid,
    pss->pbufwrite);
 }
 (void) checkpoint(dbt, (XLRMARKER *) NULL,
  DBCKPT_DOCHECKPOINT | DBCKPT_REBOOT);
 /*
 ** Print message to errorlog to indicate the
 ** end of checkpoint.
 */
 mnt_ex_print(EX_NUMBER(RECOVER2, REC_CKPT_END), EX_INFO, 1,
   dbt->dbt_dbnlen,
   dbt->dbt_dbname);
 if (TRACECMDLINE(RECOVER, 57) &&
   (local_rectype == REC_INIT))
```

```
    {
     scerrlog("After checkpointing '%.*s', the number of writes done by this thread %d is %d.\n",
       dbt->dbt_dbnlen,
       dbt->dbt_dbname,
       pss->pspid,
       pss->pbufwrite);
    }
   }
#if B1
   /*
   ** For B1 Secure SQL Server, we must reinitialize
   ** the high-SLID for the database, since recovery
   ** may have redone inserts to syslabels from the log.
   */
   (void) slm_set_highslid(dbt);
   /*
   ** Perform a label consistency check on the database
   ** just recovered unless the database is master. In
   ** this case, the server will be shut down anyway and
   ** the subsequent reboot will cause a label consistency
   ** check of all recoverable databases.
   */
   if (dbt->dbt_dbid != MASTERDBID)
   {
    if (!slm_label_consistency_check(dbt->dbt_dbid,
       DBCC_NOREPORT, DBCC_NOFIX))
    {
     ex_callprint(EX_NUMBER(RECOVER, REC_DB_INCONS_SECLAB), EX_INFO, 2,
          dbid);
    }
   }
#endif /* B1 */
  }
 }
 /*
 ** If we've recovered a LOAD TRANSACTION, the database must now
 ** be tidied with methods that only a load xact module would know.
 ** The called function's API specifies an XDES with an open read-only
 ** xls session, a criterion which this xdes indeed satisfies.
 */
 if (local_rectype == REC_LDXACT)
 {
  ldx_markendrec(xdes);
 }
 /*
 ** Do all necessary cleanup/house-keeping operation
 ** before leaving this routine.
 **
 **  Note:-
 **  Only activities that are permissible on
**  the database whether the undo-pass is run
**  or not is allowed here.
**
**  Any code (activities) present here will be
**  executed during boot-time and load-time recovery.
*/
/*
** If recovering master, recovery has now progressed
** to the point that sysmessages is usable.
*/
if (dbid == MASTERDBID)
{
 Resource->rflag1 &= ~R_BEFOREMASTER;
 /* Print quiesce db message if there was one. */
 if (quiescedb_msg_params.message_number)
 {
  /*
  ** All quiesce db messages contain the single
  ** '%.*s' variable.
  */
  ex_callprint(quiescedb_msg_params.message_number,
        quiescedb_msg_params.message_severity,
        quiescedb_msg_params.message_state,
        dbt->dbt_dbnlen, dbt->dbt_dbname);
 }
}
/* Get dbinfo again in order to update it.  */
ITAGINIT(itagp);
dbiptr = ind_dbinfoget(logsdes, itagp);
/*
** If we crashed during a dump with no_log, one or both of the
** following fields will be non-zero. Clear them now that
** recovery has successfully finished.
*/
dbiptr->dbi_pretruncpg = 0;
dbiptr->dbi_posttruncpg = 0;
/* Release dbinfo structure, flushing changes to disk. */
ITAGDIRTY(itagp);
ITAGWRITE(itagp);
ind_dbinforelease(itagp);
 /*
 ** write the updated suspect list to disk
 */
 if (SG_CHECK(dbt) && !sg_writedisk_suspectinfo(dbt))
  ex_raise(RECOVER, UT_RETURN, EX_CONTROL, 32);
/*
** Copy the duplicate logptr to Sysdatabases.
**
** If the database has gone through both redo-pass and undo-pass
** then there is no problem in copying the log pointer.
```

```
**
** Copying the duplicate logptr to sysdatabase is necessary here
** even though we have not done the undo-pass of the recovery,
** because failure to do so will have some strange side-effects
** if the database goes down in the middle and later on 'dump tran
** with no_truncate' is done.
**
** Consider the following sequence of operation,
**
**  Dump database    (where logfirst = 100 and seq.no = 9.05 AM)
**  Dump transaction (where logfirst = 100 and seq.no = 9.10 AM)
**  Dump transaction (where logfirst = 200 and seq.no = 9.15 AM)
**
** Assume that we are not copying the log pointer, during load
** this would the content of sysdatabases catalog,
**
**  load database (in sysdatabases: logfirst = 100, seq.no = 9.05AM)
**  load tran-1   (in sysdatabases: logfirst = 100, seq.no = 9.05AM)
**  load tran-2   (in sysdatabases: logfirst = 100, seq.no = 9.05AM)
**
** Now if the SQL server crashes and the device on which the
** loaded database present is inaccessible then the only alternative
** for us is doing a 'dump tran with no_truncate'. Doing so will try
** to dump the log chain starting from page 100 instead of the actual
** start of the log at page 200. This could lead to some strange
** results.
**
** It is not a problem to copy the logptr at the end of redo-pass
** of recovery instead of doing it after the database has been
** completely recovered (ie., undo-pass was also run). The reason
** is, before copying the logptr to sysdatabases all the dirty buffers
** would have been flushed to disk (see ldx_markendrec routine).
** So even if the server crashes the changes prior to oldest active
** tranasction (all of which are part of completed transactions) would
** be present in the database before any crash had happened to that
** database. So copying this pointer at this point and thereby moving
** the logfirst pointer would not cause any ill-effects.
*/
copylogptr(dbt);
if (TRACECMDLINE(RECOVER,7) && !(TRACECMDLINE(RECOVER,8))
      && masterupgradedone)
 {
  /* Force a shutdown after the upgrade of master */
 ueshutdown(0);
}
if ((!in_loadseq) || (pre_clr_log))
{
 /*
 ** Fill up the freespace information details either by reading
 ** from the system catalogs or by scanning the database space.
 */
 copy.check_freespace = FALSE;
 mnt_ex_print(EX_NUMBER(RECOVER2, REC_FILL_FREESPACE_START), EX_INFO, 1,
      dbt->dbt_dbnlen,
      dbt->dbt_dbname);
 th_fill_freespaceinfo(dbt, dmap_high_lpage(DBT_DISKMAP(dbt)),
  0, NOT_A_PAGE_COUNT);
 mnt_ex_print(EX_NUMBER(RECOVER2, REC_FILL_FREESPACE_END), EX_INFO, 1,
      dbt->dbt_dbnlen,
      dbt->dbt_dbname);
}
/*
** Re-reserve the clr space for push prepared transactions after the
** free space information has been initialized
*/
if (xtable->clrspacersvd)
{
 if (th_log_lct_reserve(dbt, xtable->clrspacersvd) != LOGRSV_OKAY)
 {
  ex_raise(RECOVER, REC_CLR_RESFAIL, EX_DBFATAL, 1);
 }
}
/*
** Destroy the transaction table.
**
** It is okay to destroy the transaction table even if we haven't
** done the undo-pass, because the routine that calls undo-pass
** at a later time will construct the transaction table also.
*/
copy.xtable = (XTABLE *) NULL;
destroy_xtable(xtable);
if (copy.fptab)
{
 copy.fptab = (REC_FP_TABLE *) NULL;
 destroy_fptable(fptab);
 /*
 ** Initialize the recovery time pfts status to default
 ** state (note: Its an assignment!) as recovery is
 ** completed on this database.
 */
 dbt->pfts_data.pfts_status = RECTIME_PFTS_STAT_INIT;
 if (TRACECMDLINE(RECOVER, 44))
 {
  dbt->pfts_data.pfts_status |= RUNTIME_PFTS_DISABLED;
 }
}
       /*
** Destroy the variable bitmap array.
**
** It is okay to destroy this map even if we haven't done the
** undo-pass, because the routine that calls undo-pass at a
** later time will construct the vbitmap array also.
*/
copy.allocbitmap = (VBITMAP *) NULL;
VBITMAPDESTROY(allocbitmap);
/* close the log */
copy.logsdes = (SDES *) NULL;
closetable(logsdes);
copy.xdes = (XDES *) NULL;
(void) xls_close(xdes, FALSE);
xact_end_session(xdes);
/* Release our working row buffer */
FREE_ROW_BUFFER(pss, copy.rowbuf);
/*
      ** Invalidate the supergam.
*/
      pg_invalidate_supergam(dbt);
/*
** It can be the case that this database was being upgraded when
** the server crashed.  If so system tables may have been
** created during recovery whose DESes are not cached in the
** DBTABLE. This causes OPEN_SYSTEM_TABLE() to abort, so
** fix it now by refreshing all the system DESes.
*/
des_refreshall(dbt);
      /*
** Refresh the supergam array. Sysgams may have changed during
** recovery, so it is important that the supergam array reflect
** the real page chain. Note that if this call fails, then
** a message will have been printed into the errorlog by
** pg_fill_supergam() and the supergam array will have been
** marked invalid (so that cached page numbers will not be used).
*/
pg_fill_supergam(dbt);
/*
** If both redo and undo pass of recovery is complete,
** then flush pages to disk and destory the buffers
** from cache. Not doing so may result in error, if they
** get used as a part of online (especially during upgrade).
*/
if ((!in_loadseq) || (pre_clr_log))
{
       bufdbtlastlog((BUF *) NULL, dbt);
 mnt_ex_print(EX_NUMBER(RECOVER2, REC_CACHE_CLEAN_START), EX_INFO, 1,
     dbt->dbt_dbnlen,
     dbt->dbt_dbname);
 BUFDBCLEAN_CACHE(dbt, DEFAULT_CACHE_ID);
 mnt_ex_print(EX_NUMBER(RECOVER2, REC_CACHE_CLEAN_END), EX_INFO, 1,
     dbt->dbt_dbnlen,
     dbt->dbt_dbname);
 }
 /*
 ** If it is a system-wide recovery, reduce the number
 ** of non-master db's to be recovered
 */
 if ((input_rectype == REC_INIT) && (pss->pcurdb != MASTERDBID))
 {
  /*
  ** Determine and decrement the appropriate
  ** (failedover/resident) offline db count
  */
  if (!(pss->pdbtable->dbt_stat3 & DBT3_FAILEDOVER_DATABASE))
  {
   SYB_ASSERT(Resource->rnum_offlinedb > 0);
   P_SPINLOCK(Resource->ha_spin);
   Resource->rnum_offlinedb --;
   V_SPINLOCK(Resource->ha_spin);
  }
  else
  {
   SYB_ASSERT(Resource->companion_info.cnum_offlinedb > 0);
   P_SPINLOCK(Resource->ha_spin);
   Resource->companion_info.cnum_offlinedb --;
   V_SPINLOCK(Resource->ha_spin);
  }
 }
 if (local_rectype == REC_INIT)
 {
  /*
  ** Enable lock caching
  */
  SET_LOCK_CACHING_OFF(pss);
 }
 CLEAR_BACKOUT_HANDLES(&pss->pbkout_rec);
 return (TRUE);
}
/*
** REC_LOGBOUNDS
**
** Description:
** Find the markers in SYSLOGS within which recovery should operate.
**
** Rec_logbounds gets the location of the checkpoint record from the
** dbinfo. It fills in marker for checkpoint record. It fetches the
** checkpoint record and fills in a copy of the checkpoint record for
** caller. Markers for first and last records to be scanned by recovery
** are filled in.
**
** Rec_logbounds deals with a checkpointless log. If recovery happens
** after a crash during a dumpxact with no_log then there may be a
** checkpointless log. In this case recovery starts from either the
** pretruncation or posttruncation first page of the log from the dbinfo.
** In this case xls_fixlastlog is also called to find the last page of the
** log.
**
** Parameters
** rectype   -- recovery token shows type of recovery
** dbt   -- pointer to DBTABLE for database
** logsdes    -- open sdes for syslogs
** ckptrec_to_fill  -- pointer to caller's space for checkpoint
**       record, filled in by this routine.
** ckpt_marker_to_fill  -- ptr to caller's space for marker of
**       checkpoint record filled in by this routine.
** first_log_marker_to_fill - ptr to caller's space for marker of
**       earliest record  of scan, filled in by
**       this routine.
** last_log_marker_to_fill -- ptr to caller's space for marker of
**       latest log record to be scanned, filled
**       in by this routine.
** quiescentpt_info -- the indicator to what to get for quiescent
**       point.
**
** Returns
** none
**
** Side Effects
** If the log is mirrored a single end of log is determined (by
** xls_fixlastlog). xls_fixlast log also updates the sysindexes row to
** have the correct last page.
**
** Supergam array is invalidated.
** History
** written 08/17/85 (cfr)
** allocation code rewritten 2.11.86  (cornelis)
** handle checkpointless log 3.28.89  (meena)
** 07/18/90 (dg) only copy first rid (only) rid in chekpoint record.
** Autumn 1996 (asherman) rename and rewrite this function
**    (formerly rec_init) for Earl recovery.
** Summer 1999 (scott) juggled logic on behalf of log space accounting.
*/
void
rec_logbounds(int rectype, DBTABLE * dbt, SDES *logsdes,
 XCKPT * ckptrec_to_fill, XLRMARKER * ckpt_marker_to_fill,
 XLRMARKER * first_log_marker_to_fill,
 XLRMARKER * last_log_marker_to_fill,
 int quiescentpt_info)
{
 DBINFO  *dbinfop; /* points to dbinfo  */
 ITAG  itagdbinfo; /* to access dbinfo  */
 XDES  *xdes;  /* to access log  */
```

```
SDES  *local_logsdes; /* local sdes opened for SYSLOGS when
    ** the xls session is opened  */
XCKPT  *xckpt;  /* to decode checkpoint record */
LOH  loh;  /* Used to fetch quiescent log rec. */
XLSCAN  *xlscan; /* Used to fetch quiescent log rec. */
LOGSCAN  logscan; /* Used to fetch quiescent log rec. */
pgid_t   firstpage; /* first page of checkpointless log */
       PERPETUAL_COUNTER dbinfo_logallocs_at_checkpoint;
uint32  pages_after_ckpt;
XLRMARKER quiescentpt_mrkr;
XLRMARKER  dbi_checkpt;
SYB_BOOLEAN quiescentpt_mrkr_moved;
    /* Has the quiescent marker moved? */
int  mass_size;
RECOVERY_INFO *recovery_info;
LOCALPSS(pss);
/* keep these backout variables in memory */
VOLATILE struct
{
 XDES *xdes;
} copy;
/* Initializations. */
copy.xdes = (XDES *) NULL;
quiescentpt_mrkr_moved = FALSE;
recovery_info = &Resource->rrecovery_info;
if (ex_handle(EX_ANY, EX_ANY, EX_ANY, rec_handle))
{
 if (copy.xdes)
 {
  xdes = copy.xdes;
  (void) xls_close(xdes, TRUE);
  copy.xdes = (XDES *) NULL;
  xact_end_session(xdes);
 }
 EX_DELETE;
 ex_raise(RECOVER, REC_RETURN, EX_CONTROL, 28);
}
/*
** Invalidate the supergam array prior to running recovery. This
** ensures that no dependence on sygams page numbers is made, since
** sysgams itself might change during recovery.
*/
pg_invalidate_supergam(dbt);
/*
** Print message to errorlog to indicate the start of
** estimating log boundaries.
*/
mnt_ex_print(EX_NUMBER(RECOVER2, REC_LOGBOUNDS_START), EX_INFO, 1,
  dbt->dbt_dbnlen, dbt->dbt_dbname);
/* Under trace flag 3457, print the read waits. */
```

```
if (TRACECMDLINE(RECOVER, 57) && (rectype == REC_INIT))
{
 scerrlog("Before estimating log boundary for database '%.*s', the data read waits is %d, log read waits is %d.\n",
  dbt->dbt_dbnlen, dbt->dbt_dbname,
  pss->pdatareadwait, pss->plogreadwait);
}
copy.xdes = xdes = xact_begin_session(dbt);
(void) xls_open(xdes, pss, XLS_READONLY, 0);
/* Set up sdes to access the log. */
local_logsdes = xdes->xxs->xxs_syslogs_scan;
/*
** If recovery type is REC_INIT, and the pool with largest i/o size
** is available, set up the log scan to use it to optimize log i/o.
** Also set up the apf scan for the log i/o.
*/
if ((rectype == REC_INIT) && (recovery_info->status &
     REC_INFO_USE_LARGEST_IO_POOL))
{
 mass_size = BYTES_FROM_KUNITS(cm_largest_mass(DEFAULT_CACHE_ID));
 if (mass_size != CFG_GETCURVAL(cmaxdbpagesize))
 {
  local_logsdes->sbufinfo = &local_logsdes->sbi_data;
  /*
  ** If we are in the process of tuning for parallel
  ** recovery, use the default pool to determine the
  ** logbounds so as not to overwhelm the I/O subsystem.
  ** Overwhelming the I/O subsystem can lead to
  ** misleading indicators regarding the capability of
  ** the I/O subsystem.
  ** Grab the spinlock, to be safe...
  */
  P_SPINLOCK(Resource->ha_spin);
  if ((recovery_info->status & REC_INFO_PARALLEL) &&
      !(recovery_info->status & REC_INFO_TUNE_COMPLETE))
  {
   V_SPINLOCK(Resource->ha_spin);
   if (TRACECMDLINE(RECOVER, 56))
   {
    scerrlog("Using the default pool as the server is in the tuning  process for recovery\n");
   }
  }
  else
  {
   V_SPINLOCK(Resource->ha_spin);
   /*
   ** When not in the tuning process  use large I/O
   ** No need to save the old value, as the
   ** logsdes will be destroyed after recovery.
   */
   SDES_SBUFINFO_VSIZE(local_logsdes) =  mass_size;
  }
  local_logsdes->sbufinfo->svs_strategy.vstrategy |=
    (VS_STRTGY_PREFETCH_SOFT);
 }
}
/*
** Check for crash during 'dump tran with nolog'.
**
** CASE 1: (dbi_posttruncpg == 0) AND (dbi_pretruncpg == 0)
** No special processing is required. If dumpxact crashed at all
** it was during the pretruncation phase.
**
** CASE 2: (dbi_posttruncpg == 0) AND (dbi_pretruncpg != 0)
**  If dumpxact managed to truncate and flush the page linkage
** and extent allocation map to disk, but before it got a chance
** to update dbi_postruncpage, then we need to use dbi_pretruncpage
** as the starting log page for recovery purposes.
**
** CASE 3: (dbi_posttruncpg != 0)
**  On the other hand, if the dumpxact managed to complete the
**  truncation, flushing the page linkage and extent alloc map
** to disk and updating dbi_postruncpage but before checkpoint
** is written and dumptran is finished, then we need to use
** dbi_posttruncpage as the starting log page for recovery.
** Once dbi_posttrucpage was updated and checkpoint is written
** we cannot use dbi_pretruncpage, as checkpoint could have
** gotten the page that was deallocated by the dumptran.
*/
ITAGINIT(&itagdbinfo);
dbinfop = ind_dbinfoget(logsdes, &itagdbinfo);
firstpage = ((dbinfop->dbi_posttruncpg != 0)
  ? dbinfop->dbi_posttruncpg : dbinfop->dbi_pretruncpg);
if (firstpage)
{
 /* Crash occurred during dump tran with nolog */
 /* End of this brief mutex */
 ind_dbinforelease(&itagdbinfo);
 /* Patch the start of the log */
 (void) xls_patchlogstart(xdes, firstpage, (pgid_t) 0);
 /*
 ** Return a marker to the start of the log.
 **
 ** Cougar II XLSRESOLVE
 ** Need to remove knowledge that markers are RIDs. Should
 ** firstpage be a 'firstmarker'?
 */
 first_log_marker_to_fill->xlrm__rid.pageid = firstpage;
 first_log_marker_to_fill->xlrm__rid.row.rnum = 0;
 /*
 ** Find the end of the log by starting from firstpage since we
 ** may have a checkpointless log. xls_fixlastlog() will fill in
 ** the marker pointed to by last_log_marker_to_fill .
 **
 ** The return value isn't useful because dbinfo log space
 ** values have been invalidated by the crash in DUMP TRAN
 ** WITH NO_LOG.  The log free space counts will have to be
 ** calculated from scratch before the database comes on line.
 */
 (void) xls_fixlastlog(xdes, firstpage, last_log_marker_to_fill);
 /*
 ** Set up an empty marker for the checkpoint log record
 ** All records between the first and last marker will then
 ** be considered for redo, in the redo pass.
 */
 xlm_setemptymarker(ckpt_marker_to_fill);
 /* clean up */
 copy.xdes = (XDES *) NULL;
 (void) xls_close(xdes, FALSE);
 xact_end_session(xdes);
 return;
}
/*
** This is regular boot-time/load recovery.  Get DBINFO values
** needed by either of the two following branches, so the dbinfo
** mutex can be freed before the pending XLS calls.
**
** Get:
** Marker of last checkpoint;
** last checkpoint marker's companion log space value.
**
** Then, release dbinfo.
*/
xlm_assignmarker(&dbinfop->dbi_checkpt, &dbi_checkpt);
LOGALLOCS_AT_CKPT(dbinfop, dbinfo_logallocs_at_checkpoint);
ind_dbinforelease(&itagdbinfo);
if (quiescentpt_info == NO_QUIESCENT_MARKER)
{
 /*
 ** Copy the checkpoint marker,
 ** and the logrecord for which it stands,
 ** into our caller's space
 */
 xlm_assignmarker(&dbi_checkpt, ckpt_marker_to_fill);
 ckpt_get_record(xdes, &dbi_checkpt, ckptrec_to_fill);
 xckpt = ckptrec_to_fill;
 /*
 ** Determine the marker to the last log record,
 ** and get the number of log pages after the
 ** checkpoint record that we pass as the starting point.
 ** Log reallocations happen later.
```

```
  */
  pages_after_ckpt
   = xls_fixlastlog(xdes,
    xlm_markertopage(&dbi_checkpt),
    last_log_marker_to_fill);
# ifdef TRACE_REC
  if (TRACECMDLINE(RECOVER, 0))
  {
   xlm_printmarker(
   "RECOVERY: Last checkpoint log record as found in scan = %s\n",
     &dbi_checkpt);
   xlm_printmarker(
   "RECOVERY: Last log record in log = %s\n",
    last_log_marker_to_fill);
  }
# endif
  /*
  ** Initialize the marker which will be used as the
  ** start point for recovery's scans.
  */
  if (xckpt->xstat & CKPT_HASACTIVE)
  {
   /*
   ** start scan at oldest active transaction pointed
   ** to by checkpoint record
   */
   xlm_assignmarker(&xckpt->xoldestlr,
      first_log_marker_to_fill);
  }
  else
  {
   /*
   ** No active transactions, and not a checkpointless
   ** log (dealt with earlier) so scan from checkpoint
   ** record.
   */
   xlm_assignmarker(&dbi_checkpt,
     first_log_marker_to_fill);
  }
 }
 else if (quiescentpt_info &
  (QUIESCENT_LASTLOG_MARKER | QUIESCENT_FIRSTLOG_MARKER))
 {
  /*
  ** Handle various intricacies related to finding the recovery
  ** logbounds when recovering a database which was previously
  ** loaded with transaction dump made for standby (pseduo
  ** replication) purposes.
  **
  ** A general note:
```

```
**
** If the previously loaded dump-image was that from pseduo
** replication dump, then dbi_checkpt will have the log marker
** corresponding to the quiescent point.
**
** This code is reached when recovery is called during
** the load sequence (which could be either load-tran
** recovery or online time recovery). In any case, the
** recovery start and end markers are picked up from the
** dbi_checkpt field and the xls_fixlastlog() call respectively.
*/
/*
** Assertion:
** We never call this routine looking for just the quiescent
** lastlog marker. We either look for both the quiescent first
** and quiescent last OR quiescent first log marker only.
*/
SYB_ASSERT((quiescentpt_info == QUIESCENT_FIRSTLOG_MARKER) ||
    (quiescentpt_info == (QUIESCENT_LASTLOG_MARKER |
     QUIESCENT_FIRSTLOG_MARKER)));
/*
**  1. Fill the checkpoint log record.
**
** Set the ckpt_marker_to_fill to empty marker.
*/
xlm_setemptymarker(ckpt_marker_to_fill);
/*
** 2. Calculate the last log page/record.
**
** This approach works equally well for either normal
** standby access load tran or for a transaction dump
** that was made using no_truncate option and
** is loaded after a dump made for standby access.
** For the former case, it'd be tempting to feed
** the dbi_nextcheckpt marker page to xls_fixlastlog(),
** so he could jump right to the last page without scanning.
** Unfortunately, we need him to perform a full scan from
** the first to last page, in order to maintain the log
** allocation count.
**
** Since this is load tran for standby access,
** "pages_after_ckpt" might actually be "pages after beginxact".
*/
pages_after_ckpt = xls_fixlastlog(xdes,
   xlm_markertopage(&dbi_checkpt),
    last_log_marker_to_fill);
/*
** Find a stopping place
** short of an unclosed begin tran record?
*/
```

```
if (quiescentpt_info & QUIESCENT_LASTLOG_MARKER)
{
 (void) xlm_setemptymarker(&quiescentpt_mrkr);
 /*
 ** The most recent transaction log loaded was
 ** originally dumped WITH STANDBY_ACCESS.
 ** Find the real quiescent point, ie the point
 ** in the log where _no_ transactions are incomplete.
 ** The dump is terminated by either a CHECKPOINT
 ** log record, or a BEGINXACT log record. While
 ** the CHECKPOINT log record is acceptable as the
 ** end marker for recovery, the BEGINXACT is not,
 ** since recovery will believe that this is an
 ** unfinished transaction. Searching back to the
 ** previous ENDXACT (or CHECKPOINT) will
 ** establish a point (the end marker for
 ** recovery), where no transactions are active.
 **
 ** In the (unfortunate) case where a non-quiescent
 ** "quiescent" point has been established for the
 ** last log marker, raise an error even though
 ** to do so will result in the database being marked
 ** suspect.
 */
 if (! rec_find_quiescent_marker(dbt,
  last_log_marker_to_fill, &quiescentpt_mrkr,
  QUIESCENT_LASTLOG_MARKER))
 {
  ex_raise(RECOVER, REC_GIVEUP, EX_DBFATAL, 6,
   dbt->dbt_dbnlen, dbt->dbt_dbname,
   dbt->dbt_dbid);
  /* not reached */
 }
 /*
 ** Check to see whether the real quiescent point
 ** differs from the apparent quiescent point.
 */
 if (xlm_cmpmarker(&quiescentpt_mrkr,
    last_log_marker_to_fill) != 0)
 {
  /*
  ** Set the end marker for recovery to
  ** the real quiescent point, and note that
  ** it moved.
  */
  quiescentpt_mrkr_moved = TRUE;
  xlm_assignmarker(&quiescentpt_mrkr,
    last_log_marker_to_fill);
 }
}
```

```
/*
** 3. Find the starting marker for recovery.
**
** During load time recovery of a pseudo replication dump,
** it is nothing but the information stored in the
** dbi_checkpt field.
**
** At this point dbi_checkpt field may or may not be a
** checkpoint record.
**
** E.g. If this the first load tran then dbi_checkpt
** would be a CHECKPOINT type log record. Otherwise
** it would be BEGINXACT type log record.
**
** If it is a checkpoint record and if it had any active
** transaction at the time of dump then we have to find
** that by looking at the oldest active transaction pointed
** to by checkpoint log record and use that as the starting
** point of the recovery.
*/
if (quiescentpt_info & QUIESCENT_FIRSTLOG_MARKER)
{
 /*
 ** If a QUIESCENT_LASTLOG_MARKER had been requested
 ** (which through assumptions made by this routine
 ** means that QUIESCENT_FIRSTLOG_MARKER _must_
 ** also have been specified), then it is possible
 ** that the end marker for recovery has been moved
 ** to the nearest ENDXACT/CHECKPOINT log record
 ** found in a backwards direction in the log.
 ** In these circumstances, care must be taken
 ** to ensure that the start marker for recovery
 ** does not come _after_ the end marker for
 ** recovery, but that it is moved to the same point
 ** as the (modified) end marker. This would happen
 ** when dbi_nextcheckpt and dbi_checkpt point to
 ** the same log record, a situation which would
 ** occur after the recovery (redo pass) of a
 ** transaction log.
 */
 if ((quiescentpt_mrkr_moved) &&
  (xlm_cmpmarker(&dbinfop->dbi_nextcheckpt,
    &dbinfop->dbi_checkpt) == 0))
 {
  /*
  ** This situation will currently only occur
  ** if an ONLINE FOR STANDBY_ACCESS
  ** has been done. Ensure that the start
  ** marker for recovery is moved backwards
  ** to the real quiescent point in the same
 ** way that the end marker was. (As a
 ** general note, moving the start marker
 ** backwards should not impact the
 ** correctness of recovery, only the
 ** amount of log that needs to be scanned).
 */
 xlm_assignmarker(&quiescentpt_mrkr,
   first_log_marker_to_fill);
}
else
{
 /*
 ** Use dbi_checkpt as the start marker for
 ** recovery.
 */
 xlm_assignmarker(&dbinfop->dbi_checkpt,
   first_log_marker_to_fill);
 /*
 ** The quiescent point should either be
 ** a BEGINXACT log record or a CHECKPOINT
 ** log record. If it is a CHECKPOINT log
 ** record, verify that it is indeed a
 ** quiescent point by checking for any
 ** recorded oldest active transaction.
 ** Transactional integrity of a database
 ** can be compromised by using a
 ** non-quiescent starting point for
 ** recovery in error. It is more
 ** difficult to prove the quiescence of
 ** of a BEGINXACT quiescent point, so this
 ** check is omitted here.
 **
 ** Start by setting up a scan to obtain
 ** the quiescent log record.
 */
 xlscan = XLSCANPTR(&logscan);
 xlscan->xlsc_mode = (XLSCAN_FORWARD |
    XLSCAN_LOW  |
    XLSCAN_HIGH);
 xlm_assignmarker(first_log_marker_to_fill,
    &xlscan->xlsc_low);
 xlm_assignmarker(first_log_marker_to_fill,
    &xlscan->xlsc_high);
 (void) xls_startscan(xdes, &logscan);
 xckpt = (XCKPT *) xls_getnext(xdes, &loh);
 /*
 ** Verify that:
 ** a) a valid log record has been returned.
 ** b) it is either a BEGINXACT or CHECKPOINT
 **    log record.
```

```
   ** c) if it is a CHECKPOINT log record, that
   **    it is quiescent, ie, no active xacts
   **    were recorded.
   */
   if ((!xckpt) ||
       ((loh.loh_op != XREC_BEGINXACT) &&
        (loh.loh_op != XREC_CHECKPOINT)) ||
       ((loh.loh_op == XREC_CHECKPOINT) &&
        (xckpt->xstat & CKPT_HASACTIVE)))
   {
    ex_callprint(EX_NUMBER(RECOVER2, REC_NO_QUIESCENT_FIRSTLOGMRKR), EX_CMDFATAL, 1,
     dbt->dbt_dbnlen,
     dbt->dbt_dbname,
     xlm_markertopage(first_log_marker_to_fill),
     xlm_markertornum(first_log_marker_to_fill));
    ex_raise(RECOVER, REC_GIVEUP, EX_DBFATAL, 4,
     dbt->dbt_dbnlen,
     dbt->dbt_dbname,
     dbt->dbt_dbid);
                              /* Not reached. */
   }
   (void) xls_endscan (xdes);
  }
 }
}
/*
** Install in dbtable the current total number of log pages
** allocated since the beginning of Time;  start with the
** value at the time the checkpoint (or beginxact) record was
** written and add the number of pages currently in the log
** after the page with that record.
**
** Once the database is on line, access to this dbtable field
** is protected by XLS.
*/
STRUCTASSIGN(dbinfo_logallocs_at_checkpoint,
      dbt->dbt_logallocs);
ADD_TO_PERPCOUNTER(dbt->dbt_logallocs, pages_after_ckpt);
/*
** If we are in parallel recovery and tuning, drive the I/O subsystem
** by reading the log between the first log marker and checkpoint
** log marker
**
** Grab the spinlock, to be safe...
*/
P_SPINLOCK(Resource->ha_spin);
if ((Resource->rrecovery_info.status & REC_INFO_PARALLEL) &&
 !(Resource->rrecovery_info.status & REC_INFO_TUNE_COMPLETE))
{
 V_SPINLOCK(Resource->ha_spin);
 rec__read_log(xdes, XLM_MARKERTOPAGE(first_log_marker_to_fill),
  XLM_MARKERTOPAGE(ckpt_marker_to_fill));
}
else
{
 V_SPINLOCK(Resource->ha_spin);
}
/*
** Print message to errorlog to indicate the end of
** estimating log boundaries.
*/
mnt_ex_print(EX_NUMBER(RECOVER2, REC_LOGBOUNDS_END), EX_INFO, 1,
  dbt->dbt_dbnlen, dbt->dbt_dbname);
if (TRACECMDLINE(RECOVER, 57) && (rectype == REC_INIT))
{
 scerrlog("Used %d buffer pool for log i/o for database '%.*s'. Until now, the thread %d had %d Data Read waits and
%d Log Read waits.\n",
   SDES_SBUFINFO_VSIZE(local_logsdes),
   dbt->dbt_dbnlen, dbt->dbt_dbname,
   pss->pspid,
   pss->pdatareadwait, pss->plogreadwait);
}
/* clean up */
copy.xdes = (XDES *) NULL;
(void) xls_close(xdes, FALSE);
xact_end_session(xdes);
/*
** If we are the last recovery thread spawned during parallel recovery
** and we are still tuning, mark the statistic to be invalid, as
** we have gone out of the acceptable zone within the sampling period
*/
P_SPINLOCK(Resource->ha_spin);
if (REC_THREAD_UNDER_INSPECTION(pss))
{
 Resource->rrecovery_info.status |=  REC_INFO_INVALID_STAT;
}
V_SPINLOCK(Resource->ha_spin);
 return;
}
/*
** REC_ANALYZE_LOG
**
** MAINTENANCE NOTE:  This function may modify the input last log
** marker *in the caller's space*, effectively truncating the work
** of recovery.  So, caller must pass address of the "live" working
** instance of this value, and keep using it.
**
** Description:
** This routine implements the analysis pass.
**
```

```
** It builds the transaction table. The analysis pass will track the
** outcome of each transaction (aborted, prepared or incomplete) and
** whether the transaction contains an incomplete nested top action.
**  The analysis pass also tracks transactions which truncated tables or
**  performed a sort.
**
** An xitem belonging to a completed transaction (committed or aborted)
**  is removed from the transaction table unless it falls into one or more
**  of the following categories:
**
**   1. It started before the recovery checkpoint and completed
**      after checkpoint. In this case restart recovery has to redo
**      all the log records logged by this transaction. This xitem
**      is retained if we are in restart recovery so that redo pass
**      knows what to do with its log records.
**
**   2. It truncated tables and aborted. In this case also the xitem
**      is retained to guide redo action. As a side effect of
**      truncating a table, the OAM page gets truncated. This
**      operation is not reversible, so it should be performed only
**      if the transaction committed. Since the number of aborted
**      transactions is much fewer than committed transactions we
**      retain the xitems for the aborted cases and truncate the
**      OAM page during redo only if the xitem is not in the table.
**
**   3. It performed a sort and aborted. Sort is a "logical" log
**      record. So redoing it requires a physical sort of the
**      rows. This operation is expensive hence it should be done
**      only if the transaction committed. Again since the number
**      of aborted transactions is much fewer than the number of
**      committed transactions, we retain the xitems of aborted
**      transactions which performed sort operations. Sort will be
**      redone only if the xitem is not found.
**
**  In addition to these, at the end of the analysis pass the transaction
**  table will contain information on each transaction that does not have
** an ENDXACT record in the current scope of recovery.
**
**  This transaction table is used during redo of some page/extent
**  deallocation related log records. The redo routines for these log
**  records mark the relevant extents with the state of the transaction
**  which performed deallocation operations on the extent. This state
**  information is used after the completion of redo pass to clean up
**  these extents.
**
** Parameters:
** rectype  - token passed to recovery indicating recovery type
** dbt  - pointer to DBTABLE for db being recovered.
** xtable  - pointer to transaction table
** firstmrkr - first record to be scanned
** lastmrkr - last record to be scanned. This may be modified
**     by this function (see "Side Effects").
**
** ckptlr  - checkpoint log record used by recovery
** analysis - used for gathering statistical information
** allocbitmap - pointer to the allocation bitmap array.
** rowbuf  - working storage for a row buffer, guaranteed
**     to be large enough to hold the largest
**     possible row.
** input_dop_status- Any contribution from caller to be added to
**     DOPARAMS.dop_status.  DOPARAMS is declared here.
**
** Side Effects:
** The parameter 'lastmrkr' may be modified if either:
** a) The analysis pass has determined that the scope of recovery
**    should be reduced.
** b) During the loading of a pre-CLR log, if a CHECKPOINT log record
**    is found which was logged during a server reboot. Multiple
**    iterations of recovery are necessary on sections of the log in
**    such cases, in order to emulate the run-time recovery sequences.
** Returns:
** none
**
** Note:
**
** This routine gets called twice in the case of load time recovery,
** once before redo-pass and once before undo-pass. If you are
** changing this code, please be aware of this fact.
**
** History:
** Autumn 1996 (asherman) written
*/
void
rec_analyze_log(int rectype, DBTABLE *dbt, XTABLE *xtable,
  REC_FP_TABLE *fptab, XLRMARKER *firstmrkr,
  XLRMARKER *lastmrkr, XLRMARKER *ckptlr,
  REC_ANALYSIS *analysis, VBITMAP *allocbitmap,
  BYTE *rowbuf, int32 input_dop_status)
{
 XDES   *xdes;   /* xdes for xls session  */
 SDES  *local_logsdes; /* the sdes opened to access log
     ** when the xls session is opened*/
 XLRMARKER curr_xlr; /* current log record marker  */
 LOH  loh;  /* Log operation header for scan */
 XRECORD  *xrec;  /* generic pointer to log  */
 XLSCAN  *xlscan; /* for log scan   */
 LOGSCAN  logscan; /* for log scan   */
 DOPARAMS doparams; /* static parameters during scan */
 int  logop;  /* Logop from log record header */
 XITEM  *xitem;  /* to access transaction table */
 int  do_result; /* return value from analysis fn */
 int  mass_size;
 int32  num_logrec_read;/* number of log records that we
    ** have processed.
    */
 LOCALPSS(pss);
 /* keep these backout variables in memory */
 VOLATILE struct
 {
  XDES *xdes;
 } copy;
 /* Initializing the backout variables */
 copy.xdes = (XDES *) NULL;
 if (ex_handle(EX_ANY, EX_ANY, EX_ANY, rec_handle))
 {
        if (copy.xdes)
        {
   xdes = copy.xdes;
   copy.xdes = (XDES *) NULL;
   (void) xls_close(xdes, TRUE);
   xact_end_session(xdes);
  }
  EX_DELETE;
  ex_raise(RECOVER, REC_RETURN, EX_CONTROL, 32);
       }
# ifdef TRACE_REC
 if (TRACECMDLINE(RECOVER, 0))
 {
  TRACEPRINT("RECOVERY: analysis pass\n");
 }
# endif
 /*
 ** Print message to errorlog to indicate the start of
 ** analysis phase.
 */
 mnt_ex_print(EX_NUMBER(RECOVER2, REC_ANALYSIS_START), EX_INFO, 1,
   dbt->dbt_dbnlen, dbt->dbt_dbname);
 /* Open a session with the logging system */
 copy.xdes = xdes = xact_begin_session(dbt);
 (void) xls_open(xdes, pss, XLS_READONLY, 0);
 /* Set up the sdes to access log. */
 local_logsdes = xdes->xxs->xxs_syslogs_scan;
 /*
 ** If recovery type is REC_INIT, and the pool with largest i/o size
 ** is available, set up the log scan to use it to optimize log i/o.
 ** Also set up the apf scan for the log i/o.
 */
 if ((rectype == REC_INIT) && (Resource->rrecovery_info.status &
     REC_INFO_USE_LARGEST_IO_POOL))
 {
```

```
 mass_size = BYTES_FROM_KUNITS(cm_largest_mass(DEFAULT_CACHE_ID));
 if (mass_size != CFG_GETCURVAL(cmaxdbpagesize))
 {
  /*
  ** Hook up sbufinfo in the sdes to the sbi_data in the
  ** sdes, although it might have been done when
  ** the sdes is initiated.
  */
  local_logsdes->sbufinfo = &local_logsdes->sbi_data;
  /*
  ** No need to save the old value, as the logsdes will
  ** be destroyed after recovery.
  */
  SDES_SBUFINFO_VSIZE(local_logsdes)= mass_size;
  local_logsdes->sbufinfo->svs_strategy.vstrategy |=
    (VS_STRTGY_PREFETCH_SOFT);
 }
}
/* Initialize DOPARAMS */
MEMZERO(&doparams, sizeof(DOPARAMS));
/* Filling only the items that assumes non-zero values */
doparams.dop_xdes = xdes;
doparams.dop_analysis = analysis;
doparams.dop_xtable = xtable;
doparams.dop_fptable = fptab;
doparams.dop_orig_rectype = rectype;
doparams.dop_rectype = rectype;
doparams.dop_allocbitmap = allocbitmap;
doparams.dop_rowbuf = rowbuf;
/* There is no need for the general purpose logsdes in this function. */
doparams.dop_logsdes = (SDES *) NULL;
/* This started as zero, but may change here. */
doparams.dop_status |= input_dop_status;
/*
** The status bit, DOP_CKPT_SEEN is used to determine whether or
** not the recovery checkpoint has been seen. DOP_CKPT_SEEN is set
** when the CHECKPOINT log record corresponding to the recovery
** checkpoint marker is processed. If an empty recovery checkpoint
** marker is passed in, then either:
** a) the server crashed during DUMP TRAN WITH NO_LOG or
** b) this dump was preceeded by the loading of a STANDBY_ACCESS
**    dump.
** In both cases, the implied recovery checkpoint is at the start
** of all the log records to be considered by recovery, so
** DOP_CKPT_SEEN is set to ensure this.
*/
if (xlm_isemptymarker(ckptlr))
{
 doparams.dop_status |= DOP_CKPT_SEEN;
}
```

```
/* extract xls scan pointer */
xlscan = XLSCANPTR(&logscan);
xlscan->xlsc_mode = (XLSCAN_FORWARD | XLSCAN_LOW | XLSCAN_HIGH);
xlm_assignmarker(firstmrkr, &xlscan->xlsc_low);
xlm_assignmarker(lastmrkr, &xlscan->xlsc_high);
(void) xls_startscan(xdes, &logscan);
num_logrec_read = 0;
while (xrec = xls_getnext(xdes, &loh))
{
 /*
 ** Check to see if this pss has been frozen. If so, call
 ** rec_freeze_thread to go to sleep.
 ** Do not use spinlock protection to check the status in pss
 ** for performance reason. The status will be checked again
 ** inside of rec_freeze_thread() under spinlock.
 */
 if (FREEZE_RECOVERY_THREAD(pss))
 {
  rec_freeze_thread();
 }
 num_logrec_read++;
 /* Get the marker for this record */
 (void) xls_getmarker(xdes, XLSM_SCAN, TRUE, &curr_xlr);
 /*
 ** If the log is corrupt in some way it is possible that the
 ** logop is bad. In this case a stack trace could result from
 ** indexing into the analysis_functions array. Instead, raise
 ** an error.
 */
 logop = loh.loh_op;
 /*
 ** For testing only:
 ** If trace flag 3459 is on, force log op to be an invalid
 ** logop, and therefore raise fatal error when processing
 ** the 100th log record for user databases with even
 ** dbid number.
 */
 if (TRACECMDLINE(RECOVER, 59) && (num_logrec_read == 100) &&
    (dbt->dbt_dbid > MODELDBID) && (dbt->dbt_dbid % 2 == 0) )
 {
  scerrlog("THREAD %d: Simulate BAD_REC error (fatal) in rec_analyze_log() by setting the logop to -1.\n",
    pss->pspid);
  logop = -1;
 }
 if ((logop < 0) || (logop > MAXRECTYPE))
  ex_raise(DO, BAD_REC, EX_DBFATAL, 12, logop);
 /*
 ** We need to mark all open transactions which preceed
 ** the recovery checkpoint as requiring redo, regardless
 ** of the recovery type. See the prologue of rec_redo_log
** for more detail.
*/
if (logop == XREC_CHECKPOINT)
{
 if (xlm_cmpmarker(&curr_xlr, ckptlr) == 0)
 {
  if (TRACECMDLINE(RECOVER, 56))
  {
   scerrlog("Seen the CKPT record in analysis pass for database '%.*s' by spid %d.\n",
     dbt->dbt_dbnlen,
     dbt->dbt_dbname,
     pss->pspid);
  }
  /*
  ** mark that ckpt has been seen so that
  ** later we can compare the position of
  ** endsort record with that of the ckpt.
  */
  doparams.dop_status |= DOP_CKPT_SEEN;
  /*
  ** Fall thru and mark all open transactions in
  ** the transaction table as requiring redo.
  */
 }
 else if ((dbt->dbt_logvers < UNDO_LOGS_CLRS_LOGVERS_ID)
      &&  (rectype == REC_LDXACT))
 {
  /*
  ** During LOAD TRAN recovery of pre-CLR
  ** logs, special consideration should be
  ** given to CHECKPOINT log records which
  ** were logged during a server reboot.
  ** Pre-11.9 recovery will roll back any
  ** incomplete transactions at this stage.
  ** In order to emulate the events at reboot
  ** time, recovery must do likewise by
  ** redefining the lower boundary of recovery
  ** to be this log record, and running the
  ** redo and undo passes up to this
  ** new lower boundary. The rest of the log
  ** is then recovered in a separate iteration.
  */
  if ((xrec->xckpt.xstat & CKPT_REBOOT)
  &&  (xlm_cmpmarker(&curr_xlr, firstmrkr) != 0))
  {
   xlm_assignmarker(&curr_xlr, lastmrkr);
   break;
  }
  else
  {
   continue;
  }
 }
 else
 {
  continue;
 }
}
/*
** For each transaction and its subordinates (for parallel
** transactions), keep a track of the last log record
** before the recovery checkpoint.
*/
if ((!REC_CKPT_SEEN(doparams.dop_status))
&&  (xitem = find_xitem(doparams.dop_xtable, &loh.loh_sid)))
{
 xfamily_put(doparams.dop_xtable, xitem,
   &loh.loh_sid, &curr_xlr);
}
/*
** Call the do-logop layer analysis routines (see analysis.c).
** These routines will construct the transaction table.
*/
do_result = (*analysis_functions[logop])(&doparams,
  xrec, &loh, &loh.loh_sid,
  &curr_xlr, 0 /* clr */);
switch (do_result)
{
  case DO_OK:
 break;
  case DO_TRUNCATE_RECOVERY:
 /*
 ** Recovery cannot proceed beyond this point in
 ** the log, but the preceding portion of the log is
 ** recoverable.  Modify the last log marker *in the
 ** caller's space* to effect a truncated log boundary
 ** for recovery.
 **
 ** Truncating the log scope of recovery can only make
 ** sense in LOAD TRAN, where there are only log records
 ** from the future; such log records have no data
 ** contemporary with them to be made consistent.
 */
 SYB_ASSERT(rectype == REC_LDXACT);
 xlm_assignmarker(&curr_xlr, lastmrkr);
 /*
 ** Now that we've changed our caller's last log marker,
 ** really make it the last log record.
 */
                      xls_patchlogend(xdes, lastmrkr, FALSE);
   /* After creating discontinuity, prevent succeeding LOAD TRANs. */
   ldx_break_sequence(xls_syslogs(xdes));
   /* No more log records should be analyzed. */
   goto end_analysis;
    default:
   SYB_ASSERT((do_result == DO_OK)
    || (do_result == DO_TRUNCATE_RECOVERY));
   break;
  }
  /* check for no more time */
  TIMESLICE_YIELD(pss);
 } /* End while (xls_getnext()) */
end_analysis:
 /*
 ** Print message to errorlog to indicate the end of
 ** analysis phase.
 */
 mnt_ex_print(EX_NUMBER(RECOVER2, REC_ANALYSIS_END), EX_INFO, 1,
     dbt->dbt_dbnlen, dbt->dbt_dbname);
 if (TRACECMDLINE(RECOVER, 57) && (rectype == REC_INIT))
 {
  scerrlog("Used %d buffer pool for log i/o for database '%.*s'. Until now, the thread %d had %d Data Read waits and
%d Log Read waits.\n",
    SDES_SBUFINFO_VSIZE(local_logsdes),
    dbt->dbt_dbnlen, dbt->dbt_dbname,
    pss->pspid,
    pss->pdatareadwait, pss->plogreadwait);
 }
 (void) xls_endscan(xdes);
 (void) xls_close(xdes, FALSE);
 copy.xdes = (XDES *) NULL;
 xact_end_session(xdes);
 /*
 ** Finally, we should mark all the xitems which belong to transactions
 ** which are incomplete. This makes things easier for redo pass by
 ** clearly labelling xactions whose log records have to be redone.
 */
 for (xitem = first_xitem(doparams.dop_xtable); xitem;
   xitem = next_xitem(doparams.dop_xtable, xitem))
 {
  if (!XITEM_TESTSTATUS(xitem, (XIT_ABORTED | XIT_COMMITTED)))
  {
   XITEM_SETSTATUS(xitem, XIT_INCOMPLETE_TRANSACTION);
  }
 }
 return;
}
/*
** REC_REDO_LOG
**
```

```
** Description:
** This routine implements the redo pass.
**
** If the redo pass is invoked at boot-time recovery, log records
** belonging to transactions that completed before recovery are
** not redone - but if any of these require post-commit cleanup
** the actions required to perform the cleanup are taken. This
** is required because a completed transaction may have deleted a
** row from a page and later deallocated the page in the same
** transaction, and committed. However, before the transaction started
** post-commit processing a checkpoint could be recorded in the log
** pointing to this transaction as the oldest active transaction. Now
** if the buffer is destroyed and the server crashes, redo will try to
** redo the transaction and fail processing the delete, because the
** timestamp on the page is before the old timestamp in the log (CR:124205)
**
**  The analysis pass leaves the transactions that started but did not
** complete before the recovery checkpoint log record, in the
**  transaction table to facilitate this. Independent of whether a
** log record is found before or after the checkpoint log record
** used by recovery, if it belongs to a transaction recorded in the
** transaction table, it is redone regardless of the status of the
** ending transaction.
**
** If the redo pass is being invoked during load tran, log records that
** have already been redone during recovery of a prior:
**    1) load database command (for the first load tran after a load
** database)
**    2) load tran command (when a transaction spans consequtive dumps)
** will not be skipped, instead they will be redone. This is to clean
** up the pages that were marked deallocated in the bitmap, but did not
** get physically deallocated because the transaction was incomplete
** during previous recovery.
**
** The reason is as the following:
**  1. For LOAD DATABASE and boottime recovery there is no guarentee that
** the change actually made it to disk at the checkpoint.
**
** 2. In theory, for LOAD TRAN recovery, the log records will have been
** redone in the previous LOAD TRAN (or LOAD DATABASE), and it will
** therefore seem unecessary to redo these log records. However:
**
** a) For LOAD TRAN recovery of pre-EARL logs, LOAD TRAN does both a redo
** pass and an undo pass (since undo will not log any CLRs). Thus the
** preceeding LOAD TRAN or LOAD DATABASE will have rolled back this
** transaction so that the changes will need to be redone.
**
** b) For LOAD TRAN of post-EARL logs, all timestamped operations for log
** records which preceed the recovery checkpoint will not need redoing.
** However, there is a requirement to redo deallocation-type log records
** in order that ex_status reflect the completion status of the
** transaction (EX_DEALL_COMPLETED_XACT or EX_DEALL_OPEN_XACT) so that
** cleanup (or retention) of the relevant bit in ex_dealloc is correctly
** done by pg__zap_dealbit_alloc() after the redo pass. A side effect of
** redoing deallocation-type log records in phase I is that all other
** allocation page work for the log record will also be redone (since
** this work is not timestamp based). This is safe, since this transaction
** only completes in phase II, so that any other allocator of the page
** will only do this in phase II, and this work is always redone.
** Note, exception to this rule is redo of sort operations which logged
** their XREC_ENDSORT prior to the CKPT record, and yet have their
** XREC_ENDXACT after the ckpt. In this case, recovery will not redo this
** sort operation. Please refer to redo_endsort() for more details.
** (cr203674-1)
**
** No extra processing is required on behalf of the transaction,
** when the ENDXACT record for the transaction is seen because all log
** records written for the transaction have been scanned and redone. Thus
** if the ending status of a transaction is ABORT, CLRs would have been
** logged for the undo of the changes in the transaction before the
** ENDXACT record itself was written, and the CLRs would have been
** processed when they were scanned.
**
**  Some deallocation related log records depend on the fact that the state
**  of the transaction (whether the transaction completed or not) is known
**  at the time of redo. This information is used to mark the extents which
**  have deallocation operations in them. After the redo pass, the extents
**  which only have deallocations done by completed transactions can be
**  cleaned up. This makes load xact recovery much easier.
**
** The routines which deal with such deallocation log records use the
**  transaction table to determine whether the transaction completed or
**  not. Therefore the transaction table has to be built before redo
**  pass commences.
**
** Parameters:
** rectype  - token passed to recovery indicating recovery type
** dbt  - pointer to DBTABLE for db being recovered.
** logsdes  - open sdes for syslogs
** xtable  - pointer to transaction table
** rectab  - recovery resource table
** fptab  - flushed pages table for PFTS lookups
** lowmrkr  - first record in log interval being recovered
** highmrkr - last record in log interval being recovered
** ckptlr  - checkpoint log record used by recovery
** allocbitmap - pointer to allocation bitmap array.
** redo_records - number of records requiring redo (from analysis pass)
** rowbuf  - working storage for a row buffer, guaranteed
**     to be large enough to hold the largest
**     possible row.
** input_dop_status- Any contribution from caller to be added to
**     DOPARAMS.dop_status.  DOPARAMS is declared here.
**
** Returns:
** number of log records that the undo pass must read
**
** Side Effects:
**
** History:
** Autumn 1996 (asherman) written
**
*/
int
rec_redo_log(int rectype, DBTABLE *dbt, SDES *logsdes,
      XTABLE *xtable, REC_FP_TABLE *fptab, RECTABLE *rectab,
      XLRMARKER *lowmrkr, XLRMARKER *highmrkr,
      XLRMARKER *ckptlr, VBITMAP *allocbitmap, int redo_records,
      BYTE *rowbuf, int32 input_dop_status)
{
 XDES   *xdes;   /* xdes for xls session  */
 SDES  *local_logsdes; /* sdes opened when xls session is
    ** opened */
 SDES  *apf_logsdes; /* sdes opened for apf scan of the log
    ** when the xls session for apf scan is
    ** opened */
 pgid_t          lastext;        /* used to facilitate pg_log_realloc */
 SDES            *alloc_sdes;    /* used to facilitate pg_log_realloc */
 XLRMARKER xlr;  /* temp. log record marker  */
 LOH  loh;  /* Log operation header for scan */
 XRECORD  *xrec;  /* generic pointer to log  */
 XLSCAN  *xlscan; /* for log scan   */
 LOGSCAN  logscan; /* for log scan   */
 DOPARAMS doparams; /* static parameters during scan */
 XITEM  *xitem;  /* for trans w/ incomplete nta */
 SDES            *dop_sdes;      /* local variable for accessing
    ** database pages */
 ITAG  itagdbinfo; /* need some logmarkers in dbinfo */
 DBINFO  *dbiptr;
 int  found_oldest_incomplete;
    /* oldest incomplete tran seen? */
 int  oldest_incomplete_num;
    /* ordinal number of record */
 int   msg_interval; /* gap between info messages */
 int  records_to_read_in_this_interval; /* Renewed from
        ** msg_interval
        */
 int  records_left; /* records left to redo  */
 int  records_done; /* records already completed */
 XDES   *apf_xdes;  /* xdes to read ahead  */
 SDES   *apf_sdes;  /* sdes to read ahead  */
 int  apf_lookahead; /* how far ahead is apf scan */
 LOGSCAN  apf_logscan; /* for apf log scan  */
 XLSCAN  *apf_xlscan; /* for apf log scan  */
 int  in_apf_scan; /* is apf scan active?  */
 LOH  apf_loh; /* loh for apf scan  */
 XRECORD  *apf_xrec; /* xrec from apf scan  */
 int  i;  /* simple counter  */
 XLSESSIONID *sidp;  /* pointer to session id in loh */
 int  use_lddb_rules; /* we're ldxact and we're in
    ** a special place in the log
    */
       XLRMARKER dumpdbs_last_lr; /* used only in ldxact */
 int  pre_clr_loadtran; /* flag: Tells us whether we are
    ** dealing with log from releases
    ** prior to the introduction of
    ** compensation log records.
    */
 int  percent_complete; /* Percentage of work done */
 int  apf_lookahead_disable_redo;
 int  mass_size;
 LOCALPSS(pss);
 /* keep these backout variables in memory */
 VOLATILE struct
 {
  XDES  *xdes;
  SDES *dop_sdes;
  SDES *alloc_sdes;
  XDES *apf_xdes;
  SDES *apf_sdes;
#if SANITY
  XLRMARKER xlr;
  XRECORD  xrec;
  LOH  loh;
#endif
 } copy;
 oldest_incomplete_num = records_left = redo_records;
 found_oldest_incomplete = FALSE;
 msg_interval = rec_msg_interval(records_left);
 in_apf_scan = TRUE;
 use_lddb_rules = FALSE;
 pre_clr_loadtran = FALSE;
 /* Initializing the backout variables */
 MEMZERO(&copy, sizeof (copy));
#if SANITY
 xlm_setemptymarker((XLRMARKER *) &copy.xlr);
#endif
 if (ex_handle(EX_ANY, EX_ANY, EX_ANY, rec_handle))
 {
#if SANITY
  if (!xlm_isemptymarker((XLRMARKER *) &copy.xlr))
  {
   TRACEPRINT("rec_redo_log: last log record read:\n");
   xlogprint((XRECORD *) &copy.xrec, (LOH *) &copy.loh,
    (XLRMARKER *) &copy.xlr, 0);
  }
#endif
        if (copy.xdes)
        {
   xdes = copy.xdes;
   copy.xdes = (XDES *) NULL;
   (void) xls_close(xdes, TRUE);
   xact_end_session(xdes);
  }
                CLOSE_SDES(&copy.alloc_sdes);
  CLOSE_SDES(&copy.dop_sdes);
        if (copy.apf_xdes)
        {
   apf_xdes = copy.apf_xdes;
   copy.apf_xdes = (XDES *) NULL;
   (void) xls_close(apf_xdes, TRUE);
   xact_end_session(apf_xdes);
  }
                CLOSE_SDES(&copy.apf_sdes);
  EX_DELETE;
  ex_raise(RECOVER, REC_RETURN, EX_CONTROL, 29);
        }
#ifdef TRACE_DO
 if (TRACE(DO, 0))
 {
  xlm_printmarker("REDO begins: starting log record = %s",
     lowmrkr);
  xlm_printmarker("ending log record = %s\n", highmrkr);
 }
#endif /* TRACE_DO */
# ifdef TRACE_REC
 if (TRACECMDLINE(RECOVER, 0))
 {
  TRACEPRINT("RECOVERY: redo pass. %d records to read\n",
    redo_records);
 }
# endif
 if (TRACECMDLINE(RECOVER, 57) && (rectype == REC_INIT))
 {
  scerrlog("Before redo starts for database '%.*s', thread %d had Data Read waits %d Log Read waits %d. The
counters will now be reset to 0.\n",
     dbt->dbt_dbnlen, dbt->dbt_dbname,
     pss->pspid, pss->pdatareadwait,
     pss->plogreadwait);
  pss->pdatareadwait = pss->plogreadwait = 0;
 }
```

```
/*
** Print message to errorlog to indicate the start of
** redo phase.
*/
mnt_ex_print(EX_NUMBER(RECOVER2, REC_REDO_START), EX_INFO, 1,
   dbt->dbt_dbnlen, dbt->dbt_dbname,
   redo_records);
/*
       ** set initial state of log reallocation variables
       **
       ** alloc_sdes - sdes for reading/writing allocation pages.
       **              This is used when reallocating log pages.
       */
       lastext = 0;
       alloc_sdes = OPEN_SYSTEM_TABLE(OPEN_SYSALLOCPG_WITH_DBT,
                              (dbid_t) UNUSED, dbt);
       copy.alloc_sdes = alloc_sdes;
/*
** TESTING POINT: If trace flag 3470 is on, raise a non-fatal
** EX_LIMIT error on user databases with even dbid number.
** Although such error will not be raised in a real situation, we
** just put this test here to test out how recovery handles
** non-fatal error.
*/
if (TRACECMDLINE(RECOVER, 70) && (dbt->dbt_dbid > MODELDBID) &&
   (dbt->dbt_dbid % 2 == 0))
{
 ex_raise(RECOVER, REC_RETURN, EX_LIMIT, 6);
}
/* Open a session with the logging system */
copy.xdes = xdes = xact_begin_session(dbt);
(void) xls_open(xdes, pss, XLS_READONLY, 0);
/* Open a session for the apf scan */
copy.apf_xdes = apf_xdes = xact_begin_session(dbt);
(void) xls_open(apf_xdes, pss, XLS_READONLY, 0);
/* Set up the sdes's for log access. */
local_logsdes = xdes->xxs->xxs_syslogs_scan;
apf_logsdes = apf_xdes->xxs->xxs_syslogs_scan;
/*
** If recovery type is REC_INIT, and the pool with largest i/o size
** is available, set up the log scan to use it to optimize log i/o.
** Also set up the apf scan for the log i/o.
*/
if ((rectype == REC_INIT) && (Resource->rrecovery_info.status &
    REC_INFO_USE_LARGEST_IO_POOL))
{
 /*
 ** Under trace flag 3457, reset the pbufwrite in Pss so that
 ** we can track the number of writes in since redo.
 */
 if (TRACECMDLINE(RECOVER, 57))
 {
  pss->pbufwrite = 0;
 }
 mass_size = BYTES_FROM_KUNITS(cm_largest_mass(DEFAULT_CACHE_ID));
 /*
 ** TESINT POINT: If traceflag 3471 is turned on, simulate a
 ** segfault for the parallel recovery thread.
 */
 if (TRACECMDLINE(RECOVER, 71) &&
   (pss->pprocess_type == RECOVERY))
 {
  scerrlog("Simulating a segfault for the parallel recovery thread\n");
  apf_logsdes = (SDES *)NULL;
 }
 if (mass_size != CFG_GETCURVAL(cmaxdbpagesize))
 {
  /*
  ** Hook up sbufinfo in the sdes to the sbi_data in the
  ** sdes, although it might have been done when
  ** the sdes is initiated.
  */
  local_logsdes->sbufinfo = &local_logsdes->sbi_data;
  apf_logsdes->sbufinfo = &apf_logsdes->sbi_data;
  /*
  ** No need to save the old value, as the logsdes will
  ** be destroyed after recovery.
  */
  SDES_SBUFINFO_VSIZE(local_logsdes)= mass_size;
  SDES_SBUFINFO_VSIZE(apf_logsdes)= mass_size;
  local_logsdes->sbufinfo->svs_strategy.vstrategy |=
    (VS_STRTGY_PREFETCH_SOFT);
  apf_logsdes->sbufinfo->svs_strategy.vstrategy |=
    (VS_STRTGY_PREFETCH_SOFT);
 }
}
/*
** open sysobjects to get an SDES which will be used in
** the lower level log op redo functions.
*/
copy.dop_sdes = OPEN_SYSTEM_TABLE(SYSOBJECTS, (dbid_t) UNUSED, dbt);
dop_sdes = copy.dop_sdes;
dop_sdes->sstat |= SS_NOCHECK;
/* open sysobjects to get an SDES for apf scan */
apf_sdes = OPEN_SYSTEM_TABLE(SYSOBJECTS, (dbid_t) UNUSED, dbt);
copy.apf_sdes = apf_sdes;
apf_sdes->sstat |= SS_NOCHECK;
apf_sdes->sbufinfo->svs_strategy.vstrategy |= VS_STRTGY_APF_NOWAIT;
/* Initialize DOPARAMS */
MEMZERO(&doparams, sizeof(DOPARAMS));
```

```
/* Filling only the items that take non-zero values */
doparams.dop_xdes = xdes;
doparams.dop_sdes = dop_sdes;
doparams.dop_xtable = xtable;
doparams.dop_rectable = rectab;
doparams.dop_fptable = fptab;
doparams.dop_orig_rectype = rectype;
doparams.dop_rectype = rectype;
doparams.dop_allocbitmap = allocbitmap;
doparams.dop_rowbuf = rowbuf;
/*
** This sdes could be used to get syslogs page down in some redo
** functions. It is important to _not_ use the local_logsdes,
** which is used for log record scan, because the scan pointer of
** the local_logsdes needs to be perserved while scanning the log.
*/
doparams.dop_logsdes = logsdes;
/* This started as zero, but may change here. */
doparams.dop_status |= input_dop_status;
/*
** Initialize IO counts.
*/
dbt->dbt_redo_numio_done = 0;
dbt->dbt_redo_numio_skipped = 0;
if (dbt->dbt_logvers < UNDO_LOGS_CLRS_LOGVERS_ID)
{
 /*
 ** Tell redo-logop routines to restore the old timestamp from
 ** log records when undoing.
 */
 doparams.dop_status |= DOP_RESTORE_OLD_TS;
}
/*
** The status bit, DOP_CKPT_SEEN is used to determine whether or not
** the recovery checkpoint has been seen, as the redo of log records
** written by transactions which complete before this checkpoint
** is not necessary - only post-commit work needs to be done for
** such transactions. DOP_CKPT_SEEN is set when the CHECKPOINT log
** record corresponding to the recovery checkpoint marker is
** processed. If an empty recovery checkpoint marker is passed in
** then either:
** a) the server crashed during DUMP TRAN WITH NO_LOG or
** b) this dump was preceeded by the loading of a STANDBY_ACCESS
**    dump.
** In both cases, all log records need to be redone, so DOP_CKPT_SEEN
** is set to ensure this.
*/
if (xlm_isemptymarker(ckptlr))
{
 doparams.dop_status |= DOP_CKPT_SEEN;
```

```
}
/* if we are LOAD TRANSACTION */
      if (rectype == REC_LDXACT)
      {
 /* Determine if we are working with a pre-11.9 log */
 pre_clr_loadtran
  = ( (dbt->dbt_logvers < UNDO_LOGS_CLRS_LOGVERS_ID)
   ? TRUE : FALSE );
 /* Initialize */
 xlm_setemptymarker(&doparams.dop_lastsortredone);
 ITAGINIT(&itagdbinfo);
 dbiptr = ind_dbinfoget(logsdes, &itagdbinfo);
 /*
 ** If a previous ldxact which crashed before completing
 ** recovery had already redone some of the sorts in the
 ** log, we will have to ensure that we dont redo those
 ** sorts again, in this later attempt at recovery.
 ** This information is maintained in dbi_lastsortredone
 ** for databases having dbinfo versions >=
 ** DBI_HAS_LASTSORTREDONE_DBIVERS_ID and later.
 ** After LOAD TRAN recovers each sort, the RID of the ENDSORT
 ** log record which triggered the sort is flushed to disk
 ** in this field. When LOAD TRAN recovery is finished
 ** altogether, dbi_lastsortredone is cleared and flushed
 ** to disk. Hence, if this field is found to be clear
 ** (the usual case), all sorts in the preceeding LOAD TRAN
 ** were completed. If it is not clear, then it indicates
 ** that there was a crash during LOAD TRAN recovery, and
 ** all sorts up to and including that  indicated by
 ** dbi_lastsortredone, have already been redone.
 */
 if ((dbiptr->dbi_version >= DBI_HAS_LASTSORTREDONE_DBIVERS_ID)
  && (!xlm_isemptymarker(&dbiptr->dbi_lastsortredone)))
 {
  /*
  ** Sort-redo needs dbi_lastsortredone to decide if
  ** a sort has to be redone. Move it to doparams.
  */
  xlm_assignmarker(&dbiptr->dbi_lastsortredone,
     &doparams.dop_lastsortredone);
  if (TRACECMDLINE(RECOVER, 25) || TRACE(LDLOG, 9))
  {
   xlm_printmarker(
    "recovery:dbi_lastsortredone=%s\n",
     &doparams.dop_lastsortredone);
  }
 }
 /*
 ** if there's a dumpdb's-last-checkpoint-rid
 ** in the dbinfo structure
 */
 if (!xlm_isemptymarker(&dbiptr->dbi_dmplastckpt))
 {
  /*
  ** Before we enter the loop to scan, determine whether
  ** the checkpoint record was also the last checkpoint
  ** record for DUMP/LOAD DATABASE.  If so, then
  ** until we reach a log record never seen by LOAD
  ** DATABASE, we must recover by its rules, not ours.
  ** That will mean passing a rectype of REC_LOADDB
  ** to the redo routines instead of REC_LDXACT.
  ** Note that we do not skip the log records between the
  ** oldest begin tran and the recovery checkpoint.
  ** This is obvious for pre-CLR logs since any
  ** incomplete transactions would have been undone
  ** at the end of the previous recovery pass. However,
  ** this is also done for CLR logs. For CLR logs,
  ** the redo pass is invoked but the undo pass is
  ** deferred till online database time, so this set
  ** of log records would have already been processed
  ** by the previous load tran or load db command.
  ** However, simply revisit them, and for the
  ** deallocation log records, mark the page in
  ** the deallocation bitmap, so that at the end of redo,
  ** if the transaction has completed, clean up these
  ** pages.
  */
  if (xlm_cmpmarker(ckptlr, &dbiptr->dbi_dmplastckpt) == 0)
  {
   use_lddb_rules = TRUE;
   /* save dump database's last log rid;
   ** that log record is the upper bound
   ** of where LOAD DATABASE's rules must
   ** be used in recovery
   */
   xlm_assignmarker(&dbiptr->dbi_dmplastlr,
       &dumpdbs_last_lr);
   /* Use load database rules for redoing log records */
   doparams.dop_rectype = REC_LOADDB;
  }
  if (TRACE(DO, 0))
  {
   xlm_printmarker("REDO : dump last checkpoint log record = %s",
     &dbiptr->dbi_dmplastckpt);
   xlm_printmarker("dump last log record = %s\n", &dumpdbs_last_lr);
  }
 }
 ind_dbinforelease(&itagdbinfo);
}
else
```

```
 {
#ifdef SANITY
  /*
  ** For versions which store the sort-redo pseudo-checkpoint
  ** in dbi_lastsortredone of DBINFO, it must be empty
  ** for anything other than  ldxact recovery.
  ** (if we crashed  during ldxact recovery,
  ** recovery() would have changed the rec_type
  ** from REC_INIT to REC_LDXACT before calling us)
  */
  ITAGINIT(&itagdbinfo);
  dbiptr = ind_dbinfoget(logsdes, &itagdbinfo);
  if (dbiptr->dbi_version >= DBI_HAS_LASTSORTREDONE_DBIVERS_ID)
  {
   SYB_ASSERT(xlm_isemptymarker(&dbiptr->dbi_lastsortredone));
  }
  ind_dbinforelease(&itagdbinfo);
#endif
 }
 /* extract xls scan pointers */
 xlscan = XLSCANPTR(&logscan);
 apf_xlscan = XLSCANPTR(&apf_logscan);
 xlscan->xlsc_mode = (XLSCAN_FORWARD | XLSCAN_LOW | XLSCAN_HIGH);
 xlm_assignmarker(lowmrkr, &xlscan->xlsc_low);
 xlm_assignmarker(highmrkr, &xlscan->xlsc_high);
 (void) xls_startscan(xdes, &logscan);
 apf_xlscan->xlsc_mode = (XLSCAN_FORWARD | XLSCAN_LOW | XLSCAN_HIGH);
 xlm_assignmarker(lowmrkr, &apf_xlscan->xlsc_low);
 xlm_assignmarker(highmrkr, &apf_xlscan->xlsc_high);
 (void) xls_startscan(apf_xdes, &apf_logscan);
 /*
 ** Let apf scan get well ahead
 */
 if (Resource->apf_lookahead_desired == 0)
 {
  Resource->apf_lookahead_desired = APF_LOOKAHEAD;
 }
 apf_lookahead = Resource->apf_lookahead_desired;
 /*
 ** Disable APF look ahead during REDO pass.
 */
 apf_lookahead_disable_redo = TRACECMDLINE(RECOVER, 45);
 if (apf_lookahead_disable_redo)
 {
  apf_lookahead = 0;
  in_apf_scan = FALSE;
  TRACEPRINT("rec_redo_log: APF Disabled\n");
 }
 for (i = 0 ; i < apf_lookahead ; i++)
 {
  apf_xrec = xls_getnext(apf_xdes, &apf_loh);
  if (apf_xrec)
  {
   if (!(apf_loh.loh_status & LHSR_DO_NOT_REDO))
    apf_logrecord(apf_xrec, &apf_loh, dbt,
      apf_sdes, fptab,
      rectype, TRUE);
  }
  else
  {
   in_apf_scan = FALSE;
   break ;
  }
 }
 /* Prefetch has a head start;  now commence the actual scan. */
 records_to_read_in_this_interval = msg_interval;
 while (xrec = xls_getnext(xdes, &loh))
 {
  /* check for no more time */
  TIMESLICE_YIELD(pss);
  /*
  ** Check to see if this pss has been frozen. If so, call
  ** rec_freeze_thread to go to sleep.
  ** Do not use spinlock protection to check the status in pss
  ** for performance reason. The status will be checked again
  ** inside of rec_freeze_thread() under spinlock.
  */
  if (FREEZE_RECOVERY_THREAD(pss))
  {
   rec_freeze_thread();
  }
  /*
  ** The sdes in doparams is opened here at this level,
  ** and will be used in the lower level log op redo
  ** functions for each log record.
  ** Here, we asserts that when the control gets back to
  ** this level, there is no buffer left kept in the
  ** skeepbuf pool of the sdes.
  */
  SYB_ASSERT(doparams.dop_sdes->sbufskept == 0);
#if SANITY
  (void) xls_getmarker(xdes, XLSM_SCAN, TRUE,
     (XLRMARKER *) &copy.xlr);
  MEMMOVE(xrec, &copy.xrec, sizeof(XRECORD));
  MEMMOVE(&loh, &copy.loh, sizeof(LOH));
#endif
  /* Get the marker for this record */
  (void) xls_getmarker(xdes, XLSM_SCAN, TRUE, &xlr);
  sidp = &loh.loh_sid;
  if (SG_CHECK(dbt))
  {
   /*
   ** Save the logrid and sessionid in ha_suspect_info
   ** area for exception handler to print the error
   ** message
   */
   SG_SUSPECT_SAVE_LOGMARKER(dbt, &xlr, sidp);
  }
  /*
  ** Reallocate the log.
  */
  lastext = xls_realloc_log(xdes, &xlr, lastext, alloc_sdes);
#ifdef TRACE_DO
  if (TRACE(DO, 0))
  {
   xlogprint(xrec, &loh, &xlr, 0);
  }
#endif /* TRACE_DO */
  /*
  ** If this is a checkpoint log record, see if it
  ** is the recovery-checkpoint record. If so, set
  ** the flag that indicates the recovery log record
  ** has been seen. All other checkpoint records can
  ** be skipped - so continue
  */
  if (loh.loh_op == XREC_CHECKPOINT)
              {
   if ((!REC_CKPT_SEEN(doparams.dop_status))
                              && (xlm_cmpmarker(&xlr, ckptlr) == 0))
                      {
    /*
    ** Indicate that the recovery checkpoint
    ** has been seen.
    */
    doparams.dop_status |= DOP_CKPT_SEEN;
                      }
                      else
                      {
                              goto next_redo_prefetch;
                      }
              }
  xitem = find_xitem(xtable, sidp);
  /*
  ** Handle completed nested top actions (NTAs). Since an
  ** NTA is itself a transaction, with the same session id
  ** as its owning transaction, care must be taken to
  ** skip NTAs which complete before the recovery
  ** checkpoint, but whose owning transaction is incomplete
  ** at the recovery checkpoint. Redoing such an NTA would
  ** cause problems with unecessarily redoing page
** deallocations, which may subsequently have been allocated
** to a transaction which commits before the recovery
** checkpoint. Since the latter will not be redone, data
** corruption will result.
**
** Determine whether this log record is part of a
** nested top action which completed before the
** recovery checkpoint. This is the case if:
** a) the log record is logged within a nested top
**    action AND
** b) the log record is seen before the recovery
**    checkpoint AND
** c) if the transaction contains an open nested top
**    action at the time of the recovery checkpoint
**    (given by xit_active_nta_at_ckpt), and it is
**    not this nested top action (given by
**    xit_start_nta). In other words, this entire
**    nested top action preceeds the recovery
**    checkpoint.
**
** BEGINTOPACTION and ENDTOPACTION log records are never
** skipped here, regardless of whether the NTA is to be
** skipped or redone. If the owning transaction itself needs
** to be redone, then the redo routines for these log records
** will set up and clear the context necessary to determine
** whether the remaining NTA log records need to be
** skipped, or not. If the owning transaction completed
** before the recovery checkpoint and therefore does
** not need to be redone, then all log records related
** to the transaction will be skipped anyway.
*/
if ((xitem)
&&  (XITEM_TESTSTATUS(xitem, XIT_LOGGED_BY_NTA))
&&  (!REC_CKPT_SEEN(doparams.dop_status))
&&  (xlm_cmpmarker(&xitem->xit_active_nta_at_ckpt,
   &xitem->xit_start_nta))
&&  (loh.loh_op != XREC_BEGINTOPACTION)
&&  (loh.loh_op != XREC_ENDTOPACTION))
{
 /* Skip processing the log record. */
 records_left--;
 goto next_redo_prefetch;
}
/*
** The analysis pass retains xitems in the xact table in four
** cases:
** 1. The transaction is open. In this case we have to
**     redo the log record.
**  2. The transaction started before the recovery
**     checkpoint. In this case also we have to redo the
**     log record.
** 3. The transaction truncated tables and aborted. In
**    this case we have to redo the log record only if the
**     transaction started before the recovery checkpoint.
**     If it completed before the recovery checkpoint its
**     changes are already on disk so we have to only redo
**     log records with post commit work.
** 4. The transaction performed a sort and aborted. The
**     sort log record is redone only during load xact
**     recovery.
**
** Note that 1 and 2 are not mutually exclusive. To further
** complicate the matters in either case the transaction could
** have truncated tables.
**
** The if condition below ensures that the log record is
** redone only if 1. or 2. or both are true.
*/
if (xitem && (XITEM_TESTSTATUS(xitem, XIT_REDO_TRANSACTION)
            || XITEM_TESTSTATUS(xitem, XIT_INCOMPLETE_TRANSACTION)))
{
 /* All redo actions must be performed for the
 ** log record because it belongs to a transaction
 ** that completed after the recovery checkpoint
 ** was seen or it did not complete at all.
 */
 doparams.dop_status &= ~DOP_REDO_POST_COMMIT;
 if (XITEM_TESTSTATUS(xitem, XIT_INCOMPLETE_TOPACTION))
 {
  /*
  ** If transaction contains an incompleted
  ** nested top action then save in the xitem a
  ** marker to the last log record seen.
  */
  xlm_assignmarker(&xlr, &xitem->xit_end_nta);
 }
 if (!found_oldest_incomplete &&
     XITEM_TESTSTATUS(xitem,
        XIT_INCOMPLETE_TRANSACTION))
 {
  /*
  ** This is the first log record
  ** of an incomplete transaction that
  ** has been seen by the redo pass.
  ** Save the ordinal number of the log record.
  */
  oldest_incomplete_num = records_left;
  found_oldest_incomplete = TRUE;
 }
}
```

```
else
{
 /*
 ** We will be here if the transaction completed.
 ** In such a case, the log record has to be
 ** redone only if the recovery checkpoint has been
 ** seen. For transactions that completed before
 ** the recovery checkpoint we only process log records
 ** that require post-commit work - all other changes
 ** should have been written to disk as a part of the
 ** checkpoint.
 ** We will not do any post-commit work if it is in
 ** transaction recovery, because we know the post
 ** commit work must have been done in the previous
 ** load tran.
 */
 if (!REC_CKPT_SEEN(doparams.dop_status))
 {
  /*
  ** Process log records
  ** that require post commit work
  */
  if (POSTCOMMIT_LOGREC(loh.loh_op) &&
   (doparams.dop_rectype != REC_LDXACT))
  {
   /* Fall thru and process the log
   ** record
   */
   doparams.dop_status |=
    DOP_REDO_POST_COMMIT;
  }
  else
  {
   /*
   ** Skip processing the log record
   ** The analysis pass retains
   ** xitems for aborted xacts
   ** which had dropextsmap log record.
   ** If this log record is an
   ** endxact log record for such
   ** a xact then remove the xitem
   ** from the table. This ensures
   ** that when the undo pass
   ** starts the table has xitems
   ** only for xacts which are
   ** open and hence require undo.
   */
   if (loh.loh_op == XREC_ENDXACT
       && xitem)
   {
      SYB_ASSERT(
      XITEM_TESTSTATUS(xitem,
       XIT_TRUNCATED_TABLE));
      rm_xitem (xtable, sidp);
     }
     goto next_redo_prefetch;
    }
   }
   else
   {
    /*
    ** After the recovery checkpoint log record is
    ** seen all log records scanned need processing.
    ** Their transaction-ids are not recorded in
    ** the transaction table, if they happened to
    ** start and complete after the recovery
    ** checkpoint. Fall thru and process log the
    ** record
    */
    doparams.dop_status &= ~DOP_REDO_POST_COMMIT;
   }
  }
  if (!(loh.loh_status & LHSR_DO_NOT_REDO))
  {
   (void) (*redo_functions[loh.loh_op])(&doparams,
    xrec, &loh, sidp, &xlr, 0 /* clr */);
  }
  else
  {
   /* No need to redo this record */
  }
  doparams.dop_status &= ~DOP_REDO_POST_COMMIT;
next_redo_prefetch:
  /*
  ** Prefetch a record and start asynch reads on referenced
  ** pages
  */
  if (in_apf_scan)
  {
   apf_xrec = xls_getnext(apf_xdes, &apf_loh);
   if (apf_xrec)
   {
    if (!(apf_loh.loh_status & LHSR_DO_NOT_REDO))
     apf_logrecord(apf_xrec, &apf_loh, dbt,
      apf_sdes, fptab,
      rectype, TRUE);
   }
   else
   {
    in_apf_scan = FALSE;
```

```
 }
}
/* Print reassuring message? */
records_left--;
if (((--records_to_read_in_this_interval) == 0)
    && (records_left > 0))
{
 /* Calculate the percentage of work done */
 records_done = redo_records - records_left;
 /*
 ** In order to avoid integer overflow that might be
 ** caused by (records_done * 100), we implicitly
 ** convert it to double, and after the calculation,
 ** the result is converted back to integer.
 */
 percent_complete = (redo_records != 0)
  ? (int) ((records_done * 100.0) / redo_records)
  : 0;
 mnt_ex_print(EX_NUMBER(RECOVER, REDO_RECS_DONE), EX_INFO, 1,
   dbt->dbt_dbnlen, dbt->dbt_dbname,
   records_done, percent_complete,
   records_left);
 records_to_read_in_this_interval = msg_interval;
 if (TRACECMDLINE(RECOVER, 57) && (rectype == REC_INIT))
 {
  scerrlog("Until now, thread %d had %d Data Read waits and %d Log Read waits for database '%.*s'. \n",
    pss->pspid,
    pss->pdatareadwait,
    pss->plogreadwait,
    dbt->dbt_dbnlen,
    dbt->dbt_dbname);
 }
}
/*
** If we are LOAD TRANSACTION but temporarily recovering
** under LOAD DATABASE rules, then ...
*/
              if (use_lddb_rules)
              {
 /*
                      ** ... if we've just seen the last log record seen
                      ** by DUMP/LOAD DATABASE, then ...
                      */
                      if (xlm_cmpmarker(&xlr, &dumpdbs_last_lr) == 0)
                      {
  /*
                              ** ...henceforth, recover under REC_LDXACT
  ** rules
  */
                              use_lddb_rules = FALSE;
    doparams.dop_rectype = REC_LDXACT;
                    }
  }
 } /* End while xls_getnext()) */
 (void) xls_endscan(xdes);
 (void) xls_close(xdes, FALSE);
 copy.xdes = (XDES *) NULL;
 xact_end_session(xdes);
 (void) xls_endscan(apf_xdes);
 (void) xls_close(apf_xdes, FALSE);
 copy.apf_xdes = (XDES *) NULL;
 xact_end_session(apf_xdes);
 CLOSE_SDES(&copy.dop_sdes);
 CLOSE_SDES(&copy.alloc_sdes);
 CLOSE_SDES(&copy.apf_sdes);
 if (TRACECMDLINE(RECOVER, 47))
 {
  TRACEPRINT("rec_redo_log: Number of IOs done [%d], skipped [%d]\n",
    dbt->dbt_redo_numio_done,
    dbt->dbt_redo_numio_skipped);
 }
 /* If we have processed any transactions, announce how many */
        if ((doparams.dop_redocount > 0) || (doparams.dop_undocount > 0))
        {
  mnt_ex_print(EX_NUMBER(RECOVER, REC_REDOPASS_NUMPROCESSED), EX_INFO, 1,
    doparams.dop_redocount, doparams.dop_undocount);
        }
 /*
 ** If we reinstantiated transactions print a  msg
 */
 if (doparams.dop_indbtcount > 0)
 {
  mnt_ex_print(EX_NUMBER(RECOVER, REC_REDOPASS_NUMREINST), EX_INFO, 1,
    doparams.dop_indbtcount);
 }
 /*
 ** Print message to errorlog to indicate the end of redo phase.
 */
 mnt_ex_print(EX_NUMBER(RECOVER2, REC_REDO_END), EX_INFO, 1,
     dbt->dbt_dbnlen, dbt->dbt_dbname);
 if (TRACECMDLINE(RECOVER, 57) && (rectype == REC_INIT))
 {
  scerrlog("REDO pass for database '%.*s' by thread %d has done %d writes. Until now, this thread had %d Data Read
waits and %d Log Read waits.\n",
    dbt->dbt_dbnlen, dbt->dbt_dbname,
    pss->pspid, pss->pbufwrite,
    pss->pdatareadwait, pss->plogreadwait);
 }
 return (oldest_incomplete_num);
}
```

```
/*
** REC__READ_LOG
**
** Read the log pages between the start page and the end page.
** This is primarily used to drive the I/O subsystem to collect I/O statistics
** for tuning during parallel recovery.
** If at any time we find that the tuning is complete, we break out and return
**
** Parameters:
**   xdes  - XDES
**  startpgno - starting page number
**  lastpgno - last page number
**
** Returns
**  void
**
** Side Effects
**  None
*/
void
rec__read_log(XDES * xdes, pgid_t startpgno, pgid_t lastpgno)
{
        XXS     *xxs;
        SDES    *sdes;
        BUF     *bp;
        pgid_t  pgno;
        LOCALPSS(pss);
 SYB_ASSERT(Resource->rrecovery_info.status & REC_INFO_PARALLEL);
 if (TRACECMDLINE(RECOVER, 56))
 {
         scerrlog("Thread %d will read log pages from %d to %d.\n",
                                pss->pspid, startpgno, lastpgno);
 }
        xxs = xdes->xxs;
        sdes = XLSI__READONLY(xxs->xxs_state) ?
                xxs->xxs_syslogs_scan : xxs->xxs_syslogs_xact;
 /*
 ** Don't do any checks during recovery
 */
        sdes->sstat |= SS_NOCHECK;
 /*
 ** Use APF scans
 */
        sdes->sbufinfo->svs_strategy.vstrategy |= VS_STRTGY_APF_SCAN;
        for (pgno = startpgno; pgno < lastpgno; pgno++)
        {
  /*
  ** If Recovery tuning is complete, break out and return
  */
  if (Resource->rrecovery_info.status & REC_INFO_TUNE_COMPLETE)
 {
  if (TRACECMDLINE(RECOVER, 56))
  {
   scerrlog("Tuning is complete. Stop reading log pages  \n");
  }
  break;
 }
               /* prepare sdes to read in next log page */
               sdes->scur.pageid = pgno;
               bp = getpage(sdes, NO_LATCH, UNUSED, (int *) UNUSED);
               bufunkeep(bp, sdes, NO_LATCH);
               TIMESLICE_YIELD(pss);
        }
        sdes->sbufinfo->svs_strategy.vstrategy &= ~VS_STRTGY_APF_SCAN;
        return;
}
/*
**  BRFINISH()
**
**  This routine clears the MASS_READING bit in the status word of the MASS
** header and wakes up any waitors on this i/o.
**
**  Parameters:
** blkioptr -- block i/o request structure, including
**  ptr to BUF structure for requested page
** status -- zero if i/o was successful
**
**  Returns:
** none.
**
**  MP Synchronization:
** Acquires and releases cache_spin lock.  Therefore it cannot be called
** at interrupt time: the call is deferred to the scheduling loop.
**
**  History:
** 3/31/86 (doughty)  written
** 06/25/86 (klwm) changed to new kernel block i/o interface
** 8/2/88 (jkr) multiprocessor changes
*/
void
brfinish(BLKIO * blkioptr, int32 status)
{
 BUF  *bp;
 SPINLOCK *cache_spin;
 BUF  *mass_ptr;
 int  incr_cntr; /* if TRUE then increment monitor */
 cacheid_t cid;
 bp = blkioptr->dbbp;
 incr_cntr = FALSE;
 mass_ptr = bp->bmass_head;
```

```
cache_spin = mass_ptr->bcache_desc->cspin;
cid = mass_ptr->bcache_desc->cid;
P_SPINLOCK(cache_spin);
if (status != 0)
{
 /*
 **  A page was not successfully read for this buffer. Unhash
 ** the buffer while under spin lock protection.
 **
 ** IMPORTANT: Any tasks which sleep waiting for the
 ** MASS_READING bit to be cleared  *must*  check for
 ** MASS_IOERR and take appropriate action.
 */
 MASS_STAT(mass_ptr) |= MASS_IOERR;
 cm_bufunhash(mass_ptr);
}
else
{
 /*
 ** Now that we have read in the page, set the MASS_LOG bit
 ** to the correct value.  no need to check bp->bmass_head
 */
 if ( bp->bpage->anp.pobjid == SYSLOGS)
  MASS_STAT(mass_ptr) |= MASS_LOG;
 else
  MASS_STAT(mass_ptr) &= ~MASS_LOG;
 MASS_STAT(mass_ptr) &= ~MASS_YIELD;
 /* Are we completing an i/o issued by APF alloc page scan */
 if (blkioptr->db_apf)
 {
  /* check the objid on the page. If the objid on
  ** the page we just read in is  0 (i.e an
  ** uninitialized page) OR the objid on the page is
  ** not matching with the objid on the SDES if
  ** SS_NOCHECK is NOT SET OR the pageno on the page
  ** is invalid then discard that mass.
  */
  if ((bp->bpage->anp.pobjid == 0) ||
    ((blkioptr->db_objid) &&
    (bp->bpage->anp.pobjid != blkioptr->db_objid)) ||
   !BP_PAGENO_OK(bp))
  {
   /*
   ** don't match - discard mass
   ** with APF_NOT_STARTED status
   */
   MASS_STAT(mass_ptr) |= MASS_APF_NOT_STARTED;
   /*
   ** If no one has yet referenced this mass, then
   ** adjust the bmass_apf counter.
   */
   if (MASS_UNREF_CNT(mass_ptr) ==
    BUFS_IN_MASS(mass_ptr->bmass_size,
          BP_PGSIZE(mass_ptr)))
    mass_ptr->bpool_desc->bmass_apf --;
   MASS_STAT(mass_ptr) &= ~MASS_READ_AHEAD;
   cm_bufunhash(mass_ptr);
   /* set to TRUE, indicate discarded mass */
   incr_cntr = TRUE;
  }
 }
}
MASS_STAT(mass_ptr) &= ~MASS_READING;
V_SPINLOCK(cache_spin);
/* deferred wakeup could happen inside spinlock critical section */
(void) upwakeup(SYB_EVENT_STRUCT(mass_ptr));
/* free blkio structure */
udfree(blkioptr);
/*
** If we are collecting statistics for recovery and the read was
** successful, increment the read counter.
** Pretest the status without spinlock.
*/
if ((Resource->rrecovery_info.status & REC_INFO_COLLECT_STAT) &&
    (status == 0))
{
 SYB_ASSERT(!(Resource->rrecovery_info.status &
    REC_INFO_TUNE_COMPLETE));
 /* check the status again under spinlock */
 P_SPINLOCK(Resource->ha_spin);
 if (Resource->rrecovery_info.status &
    REC_INFO_COLLECT_STAT)
 {
  Resource->rrecovery_info.rec_order_info->read_counter++;
 }
 V_SPINLOCK(Resource->ha_spin);
}
/* count the number of times we discard an APF read */
if (incr_cntr == TRUE)
 MONITOR_INC(mc_buffer(mass_ptr->bcache_desc,
    apf_brfinish_discard));
}
```